

HTML

Frontend · most asked

HTML interview questions

25 questions · 14 easy · 9 medium · 2 hard. Open one to see the answer, a code example and the likely follow-up, then mark it as known or to review.

1

What does `<!DOCTYPE html>` do? What happens if you leave it out?

Easy



#basics #rendering

The doctype is the first line of the document. It tells the browser to render the page in **standards mode**, following the current HTML and CSS specs.

Without it, browsers fall back to **quirks mode**, which imitates old browser bugs from the 1990s for backwards compatibility. Layout can then differ between browsers: for example, the box model, table sizing and some percentage heights behave differently.

- `<!DOCTYPE html>` is the only doctype you need for HTML5
- it is not a tag, it is an instruction to the parser
- it is case-insensitive

? Likely follow-up: How can you check which mode a page is in? (`document.compatMode`)

2

What is semantic HTML and why does it matter?

Easy



#semantics #accessibility

Semantic HTML means using elements that describe the **meaning** of the content, not just how it looks: `<header>`, `<nav>`, `<main>`, `<article>`, `<section>`, `<aside>`, `<footer>`, `<button>`, `<h1>` – `<h6>`, instead of a pile of `<div>`s.

Why it matters:

- **accessibility**: screen readers use landmarks and headings to navigate, and native elements come with **keyboard support and roles for free**
- **SEO**: search engines understand the structure and the main content
- **maintainability**: the markup is easier to read

A `<div onclick>` pretending to be a button is the classic anti-pattern: it is not focusable and does not respond to Enter or Space.

Example

```
<header>
  <nav aria-label="Main">
    <a href="/">Home</a>
  </nav>
</header>
<main>
  <article>
    <h1>Post title</h1>
    <p>Content...</p>
  </article>
  <aside>Related links</aside>
</main>
<footer>© 2026</footer>
```

? Likely follow-up: When would you use `<section>` vs `<article>` vs `<div>`?

3

What is the difference between block, inline and inline-block elements?

Easy



#basics #layout

Block elements (`<div>`, `<p>`, `<h1>`, `<ul>`, `<section>`) start on a new line and take the full width available. Width, height, margin and padding all apply.

Inline elements (`<span>`, `<a>`, `<strong>`, `<em>`) flow inside a line of text and are only as wide as their content. **Width and height are ignored**, and vertical margins do not push other lines away.

Inline-block elements (like `<button>` and `<input>`) sit in the line like inline but respects width, height and vertical spacing.

The default comes from the browser stylesheet; CSS `display` can change any element.

4

What is the difference between `<script>`, `<script async>` and `<script defer>`?

Medium



#performance #scripts

A plain `<script>` **blocks HTML parsing**: the parser stops, downloads the file, runs it, then continues.

- `async`: downloads in parallel and runs **as soon as it arrives**, pausing parsing at that moment. Order between async scripts is not guaranteed. Good for independent scripts like analytics.
- `defer`: downloads in parallel and runs **after parsing finishes**, just before `DOMContentLoaded`, in document order. Good for your app code that needs the DOM.

`type="module"` scripts are deferred by default. Putting a plain script at the end of `<body>` is the older way to get similar behaviour.

Example

```
<head>
  <!-- runs whenever it arrives, any order -->
  <script async src="analytics.js"></script>
  <!-- runs after parsing, in order -->
  <script defer src="vendor.js"></script>
  <script defer src="app.js"></script>
</head>
```

? Likely follow-up: What is the difference between `DOMContentLoaded` and `load`?

5

Compare `localStorage`, `sessionStorage` and cookies.

Easy



#storage #security

- **localStorage**: about 5 MB per origin, persists until cleared, never sent to the server
- **sessionStorage**: same API, but scoped to one tab and cleared when that tab closes
- **cookies**: about 4 KB, can have an expiry, and are **sent with every HTTP request** to their domain

Both storage APIs are synchronous and string-only, so you store objects with `JSON.stringify`. They are readable by any script on the page, so **never put auth tokens there** if you can avoid it: an XSS bug can steal them.

For sessions, prefer an `HttpOnly` cookie, which JavaScript cannot read. For larger or structured client data, use IndexedDB.

? Likely follow-up: What do the `HttpOnly`, `Secure` and `SameSite` cookie flags do?

6

What do the `HttpOnly`, `Secure` and `SameSite` cookie attributes do?

Medium 🔥🔥 #storage #security

- `HttpOnly`: JavaScript cannot read the cookie (`document.cookie` hides it), which protects session cookies from **XSS** theft
- `Secure`: the cookie is only sent over HTTPS
- `SameSite`: controls whether the cookie is sent on cross-site requests, the main defence against **CSRF**

`SameSite` values: `Strict` (never cross-site), `Lax` (sent on top-level navigation like clicking a link, the browser default) and `None` (always sent, requires `Secure`).

A good session cookie is `HttpOnly; Secure; SameSite=Lax` plus a sensible `Max-Age`.

7

How do you make a page accessible? When should you use ARIA?

Medium 🔥🔥 #accessibility #aria

Start with **native semantic HTML**, it covers most needs:

- real `<button>`, `<a href>`, `<label>` and form controls
- a logical heading order and landmarks (`<main>`, `<nav>`)
- `alt` text on images, enough colour contrast, visible focus styles
- everything usable with the keyboard alone

ARIA (`role`, `aria-label`, `aria-expanded`, `aria-live`...) adds meaning for assistive tech when HTML has no native element, for example a custom tabs widget or a live status message. It changes only what screen readers announce, not behaviour.

The first rule of ARIA: **no ARIA is better than bad ARIA**. If a native element exists, use it.

Example

```
<button aria-expanded="false" aria-controls="menu">Menu</button>
<ul id="menu" hidden>...</ul>

<button aria-label="Close dialog">x</button>

<div role="status" aria-live="polite">Saved!</div>
```

? Likely follow-up: How would you test a page for accessibility?

8

How does form validation work in HTML?

Easy



#forms #validation

HTML has **built-in constraint validation**. Attributes such as `required`, `type="email"`, `min` / `max`, `minlength` / `maxlength` and `pattern` make the browser block submission and show a message.

Good form basics:

- every input has a `<label for>` (or is wrapped in one)
- use the right `type` (`email`, `tel`, `number`, `date`) for mobile keyboards
- `name` attributes, since that is what gets submitted
- `autocomplete` hints such as `autocomplete="email"`

From JavaScript you can use `checkValidity()`, `setCustomValidity()` and the `:invalid` CSS pseudo-class.

Client-side validation is only for user experience: **always validate again on the server.**

Example

```
<form action="/signup" method="post">
  <label for="email">Email</label>
  <input id="email" name="email" type="email" required
autocomplete="email">

  <label for="pin">PIN</label>
  <input id="pin" name="pin" inputmode="numeric" pattern="[0-9]{4}"
required>

  <button type="submit">Sign up</button>
</form>
```

? Likely follow-up: What is the difference between GET and POST form submission?

9

What does the `<meta name="viewport">` tag do?

Easy



#responsive #meta

Without it, mobile browsers assume the page was built for desktop and render it in a virtual viewport about 980px wide, then zoom out. Your media queries never match and the text is tiny.

`<meta name="viewport" content="width=device-width, initial-scale=1">` tells the browser to make the layout viewport equal to the device width at 100% zoom, so responsive CSS works.

Avoid `user-scalable=no` or `maximum-scale=1`: blocking pinch-zoom is an accessibility problem.

10

Which HTML features matter for SEO?

Easy



#seo #meta

- a unique, descriptive `<title>` and `<meta name="description">` per page
- **one** `<h1>` and a logical heading hierarchy
- semantic structure and real links (`<a href>`, not click handlers on divs)
- `alt` text on meaningful images
- `<link rel="canonical">` to avoid duplicate-content issues

- Open Graph tags (`og:title` , `og:image`) for link previews on social media
- `lang` on `<html>` and structured data (JSON-LD) for rich results

Content that only appears after client-side JavaScript runs may be indexed late or not at all, which is why **server-side rendering** helps SEO.

? Likely follow-up: Why do frameworks like Next.js help with SEO?

11

What is the difference between `id` and `class` ?

Easy



#basics #selectors

An `id` must be **unique** in the document and identifies one element. It is used for fragment links (`#pricing`), `<label for>` , ARIA references and `getElementById` .

A `class` can be shared by many elements, and an element can have several classes. It is the normal hook for styling and for selecting groups of elements.

In CSS, an `id` selector has much higher **specificity** than a class, which makes it hard to override. That is why most teams **style with classes and keep ids for behaviour and linking**.

12

What are `data-*` attributes and when would you use them?

Easy



#attributes #dom

`data-*` attributes let you store custom data on any element in valid HTML, for example `data-user-id="42"` .

In JavaScript you read them through `element.dataset` , where the name is converted to camelCase: `data-user-id` becomes `dataset.userId` . Values are always strings. In CSS you can select on them (`[data-state="open"]`) or show them with `attr()` .

Typical uses: passing an id to a click handler in event delegation, UI state hooks for styling, and test selectors (`data-testid`).

Do not store sensitive or large data there: it is **visible to anyone** in the page source.

Example

```
<button class="delete" data-user-id="42" data-  
state="idle">Delete</button>  
  
<script>  
  document.addEventListener('click', (e) => {  
    const btn = e.target.closest('.delete')  
    if (btn) console.log(btn.dataset.userId) // "42"  
  })  
</script>
```

13

Why is the `alt` attribute important, and when should it be empty?

Easy



#accessibility #images

`alt` is the text alternative for an image. Screen readers read it, it shows when the image fails to load, and search engines use it.

Write it to convey the **purpose** of the image in context, not "image of...". For a linked logo, the alt should describe the destination, like "Acme home".

Use an empty `alt=""` for **purely decorative images**, so screen readers skip them.

Leaving `alt` out entirely is different and worse: many screen readers then read the file name.

Also set `width` and `height` on images so the browser reserves space and avoids layout shift.

14

How do `srcset`, `sizes` and `<picture>` work?

Medium



#images #responsive #performance

They serve the right image for the device, so phones do not download desktop-sized files.

- `srcset` + `sizes` on `<img>`: you list the same image at several widths (`400w`, `800w`) and say how wide it will be displayed. The **browser picks** the best file based on screen width and pixel density. This is **resolution switching**.
- `<picture>` with `<source>` elements: **you** decide, using media queries or formats. Use it for **art direction** (a different crop on mobile) or modern formats like AVIF/WebP with a fallback.

The inner `<img>` is always required: it is what actually renders, and it carries `alt`.

Example

```


<picture>
  <source type="image/avif" srcset="hero.avif">
  <source media="(max-width: 600px)" srcset="hero-square.jpg">
  
</picture>
```

15

How do you lazy-load images and iframes?

Easy



#performance #images

Add `loading="lazy"` to `<img>` or `<iframe>`. The browser then delays the download until the element is close to the viewport, which saves bandwidth and speeds up the initial load.

Rules of thumb:

- do not lazy-load above-the-fold images, especially the LCP hero image: it delays the most important paint. For that image use `fetchpriority="high"` instead
- always set `width` and `height` (or `aspect-ratio` in CSS) so the placeholder space is reserved and nothing jumps

Before native support, this was done with `IntersectionObserver`, which is still useful for lazy-loading other things like components or infinite scroll.

16

What is an `<iframe>`, and what are the security concerns?

Medium



#security #embedding

An `<iframe>` embeds another HTML document inside your page, for example a YouTube video, a map or a payment form. It has its own window, DOM and JavaScript context.

Security:

- the **same-origin policy** stops scripts in one frame reading another origin's DOM. Communicate with `postMessage` and always check `event.origin`

- the `sandbox` attribute restricts what embedded content can do (scripts, forms, popups) unless you allow it
- **clickjacking**: an attacker frames your site invisibly. Prevent it with the `frame-ancestors` CSP directive or the `X-Frame-Options` header

Iframes also cost performance, so lazy-load them.

🔗 Likely follow-up: How does `postMessage` work between a page and an iframe?

17

What did HTML5 add compared to older HTML?

Easy



#basics

The main additions:

- **semantic elements**: `<header>`, `<nav>`, `<main>`, `<article>`, `<section>`, `<footer>`
- **media**: native `<video>`, `<audio>` and `<canvas>`, so no Flash plugins
- **forms**: new input types (`email`, `date`, `range`) and built-in validation
- **APIs**: Web Storage, Geolocation, Web Workers, WebSockets, History API, drag and drop
- the short `<!DOCTYPE html>`

HTML is now a "living standard" maintained by WHATWG, so the term HTML5 mostly means **modern HTML** rather than a fixed version.

18

Why does a `<button>` inside a form submit it unexpectedly?

Easy



#forms

A `<button>` without a `type` defaults to `type="submit"` when it is inside a form.

Clicking a "Show password" or "Add row" button then submits the form and reloads the page.

The three types:

- `submit` (default): submits the form
- `button`: does nothing by itself, for JavaScript handlers
- `reset`: resets fields to their initial values (rarely a good idea)

Always write `type="button"` for non-submit buttons. Also prefer handling the form's `submit` event over the button's `click`, so pressing Enter in a field works too.

19

What does `tabindex` do? What values should you use?

Medium



#accessibility #keyboard

`tabindex` controls whether an element can receive keyboard focus and where it sits in the Tab order.

- `tabindex="0"`: makes a non-interactive element focusable in the normal document order (for a custom widget)
- `tabindex="-1"`: focusable only from JavaScript with `.focus()`, not with Tab. Useful for moving focus to a dialog or an error summary
- **positive values** (`1`, `2`...): force a custom order. **Avoid them**, they break the natural order and are hard to maintain

Native links, buttons and inputs are already focusable, so they need no `tabindex`. Never remove focus outlines without a visible replacement (`:focus-visible`).

20

What is the difference between `innerHTML`, `textContent` and `innerText`? Why is `innerHTML` risky?

Medium



#security #dom

- `innerHTML` parses the string as **HTML** and builds elements from it
- `textContent` sets or reads plain text; any tags are shown literally
- `innerText` is like `textContent` but layout-aware: it respects CSS (hidden text is skipped), so reading it triggers a reflow

Setting `innerHTML` with user input is a classic **XSS** hole: `<img src=x onerror="steal()">` will run. Use `textContent` for **user data**, or sanitize with a library like DOMPurify when you really need HTML.

Frameworks escape by default; React's `dangerouslySetInnerHTML` is named that way on purpose.

Example

```
const escapeHtml = (s) =>
  s.replace(/&/g, '&amp;').replace(/</g, '&lt;').replace(/>/g, '&gt;')
  .replace(/"/g, '&quot;').replace(/'/g, '&#39;')

const comment = '<img src=x onerror="alert(1)">'
console.log(escapeHtml(comment))
// &lt;img src=x onerror=&quot;alert(1)&quot;&gt; (shown as text, never runs)
```

21

Explain the critical rendering path. What blocks rendering?

Hard 🔥🔥 #performance #rendering

The steps a browser takes from HTML to pixels:

1. parse HTML into the **DOM**
2. parse CSS into the **CSSOM**
3. combine them into the **render tree** (only visible nodes)
4. **layout**: compute sizes and positions
5. **paint** and **composite** layers to the screen

CSS is **render-blocking**: the browser will not paint until the CSSOM is ready. Plain

`<script>` tags are **parser-blocking**, and they also wait for preceding CSS.

To speed it up: inline critical CSS, `defer` scripts, keep CSS small, `preload` key fonts and images, and reduce the number of round trips.

🔗 Likely follow-up: What happens when you type a URL and press Enter?

22

What are `preload`, `prefetch` and `preconnect`?

Medium 🔥👉 #performance

They are `<link rel=` resource hints that help the browser fetch things earlier.

- `preload`: fetch a resource needed **for this page** with high priority, before the parser would find it. Typical for web fonts and the hero image. Requires `as`.
- `prefetch`: low-priority fetch of something the **next page will probably need**, stored in the cache.

- `preconnect`: open the DNS + TCP + TLS connection to another origin early (for example a font CDN or API).
- `dns-prefetch`: only the DNS lookup, a cheaper fallback.

Use them sparingly: preloading too much competes with critical resources.

Example

```
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
<link rel="preload" href="/fonts/inter.woff2" as="font" type="font/woff2"
crossorigin>
<link rel="prefetch" href="/checkout.js">
```

23

How would you build a modal dialog accessibly?

Medium



#accessibility #components

Use the native `<dialog>` element. Opening it with `showModal()` gives you most of the hard parts for free:

- it renders in the **top layer**, above everything, with a `::backdrop`
- the rest of the page becomes **inert**, so focus stays inside
- Escape closes it, and focus returns to the element that opened it

Still give it an accessible name (`aria-labelledby` pointing to its heading) and a visible close button.

Building one from `<div>`s means doing all this by hand: `role="dialog"`, `aria-modal`, a focus trap, Escape handling and restoring focus.

Example

```
<button onclick="dlg.showModal()">Delete account</button>

<dialog id="dlg" aria-labelledby="dlg-title">
  <h2 id="dlg-title">Are you sure?</h2>
  <form method="dialog">
    <button value="cancel">Cancel</button>
    <button value="ok">Delete</button>
  </form>
</dialog>
```

24

What is the DOM, and how is it different from the HTML source?

Easy



#dom #basics

The HTML file is just text. The **DOM** (Document Object Model) is the **live tree of objects** the browser builds from it, which JavaScript can read and change.

They can differ:

- the parser fixes invalid markup (for example it adds a missing `<tbody>` or closes tags)
- scripts add, remove and change nodes after load
- "View source" shows the original HTML; DevTools' Elements panel shows the current DOM

In React and similar frameworks, the **virtual DOM** is a separate JavaScript copy used to compute the minimal changes to apply to the real DOM.

25

What are Web Components and the Shadow DOM?

Hard



#components #dom

Web Components are browser-native reusable components, built from three standards:

- **Custom Elements:** define your own tag, like `<user-card>`, with a class extending `HTMLElement`
- **Shadow DOM:** an encapsulated DOM subtree whose styles and markup are **isolated from the rest of the page**
- **Templates and slots** (`<template>`, `<slot>`): inert markup to clone, and placeholders for content passed in

Their strength is framework independence (design systems that work in React, Vue or plain HTML). Downsides: server-side rendering, forms and theming across the shadow boundary are harder, and most apps still use a framework's component model.