Module 12 of 16

The DOM and the browser

The DOM tree, selection, attributes versus properties and safe content; event propagation and delegation; fetch, URLs, CORS and timeouts; storage, cookies, history and observers; the rendering pipeline, loading path, vitals, XSS and CSRF.

In the browser, JavaScript's job is to read and change a tree the user is looking at, respond to what they do, talk to servers, and remember things between visits — while staying fast enough that nothing feels slow and careful enough that nobody else's script ever runs in your users' sessions. This module is the platform side of front-end work: the APIs, the rules behind them, and the two failure modes (slow and unsafe) with their causes and defences.

The DOM lesson covers the tree, selection, attributes versus properties, `textContent` versus `innerHTML`, and the reflow cost model. Events explains capture, target and bubble, `target` versus `currentTarget`, delegation with `closest`, and why `stopPropagation` is usually a coupling smell. The network lesson treats `fetch` honestly — a 404 resolves, bodies read once, nothing times out by itself, CORS is the server's decision enforced by the browser. Storage and history cover `localStorage`, the cookie flags that defend sessions, the History API behind every single-page router, and the observer APIs that replaced polling. The last lesson is rendering performance (style → layout → paint → composite, the loading path, the vitals) and the front-end threat model: XSS in layers, CSRF and `SameSite`, clickjacking, third-party scripts.

The judge has no browser, so the exercises model the mechanisms in Node with exactly the browser's rules: a selector engine over a tree, an input element with the dirty-value flag, a three-phase event dispatcher, delegation with `closest`, a `fetch` client with timeouts and an error body, real `URL` / `URLSearchParams`, a Web Storage model with quota and cross-tab events, a cookie jar that applies `Secure` / `SameSite` / `Path`, a layout-thrash counter, an HTML sanitiser plus a CSP checker — and then a virtual-DOM differ, a client-side router with a history stack, and a stale-while-revalidate cache.

CONTENTS

1	The DOM — the tree behind the page, and how to change it safely	13 min
2	Events — propagation, delegation and the event object	13 min
3	fetch, URLs, CORS and talking to servers	13 min
4	Storage, cookies, history and the browser APIs worth knowing by name	12 min
5	Rendering performance and front-end security — the two things a page must not get wrong	13 min
✓	Checkpoint — The DOM and the browser	24 min

Exercises in this module compile on Node 16. Reference solutions and the answer key follow each lesson.

LESSON 1 · 13 MIN

The DOM — the tree behind the page, and how to change it safely

The browser parses HTML into a tree of objects — the **Document Object Model** — and hands JavaScript a live handle to it: `document`. Everything a page does after it loads is a DOM operation: find a node, read or change it, create or remove one, and let the browser repaint. The API is large but shaped by a few ideas — nodes versus elements, attributes versus properties, live versus static collections, and the one method (`innerHTML`) that is both the most convenient and the most dangerous. This lesson covers the tree, selection, reading and writing, creation and removal, the reflow cost model, and the escaping rule that keeps user data from becoming code.

Nodes and elements

Every part of the document is a `Node`: elements (`<div>`), text nodes (the words between tags — whitespace included), comments, and the document itself. **Elements** are the nodes with tags, attributes and children; the text inside them is a separate child node. `Node.nodeType` (1 element, 3 text, 9 document), `parentNode`, `childNodes` (all nodes) versus `children` (elements only), `firstElementChild`, `nextElementSibling`. Most code works at the element level and treats text as a property of its parent (`textContent`).

Selecting

```
document.getElementById("app");                    // fastest for ids
document.querySelector("#app .item.active");       // first match of any CSS s
document.querySelectorAll("li[data-id]");          // ALL matches — a static N
element.querySelector("…");                        // scoped to that subtree
element.closest("[data-action]");                  // nearest ancestor (or sel
element.matches(".item");                          // does this element match?
document.getElementsByClassName("x");              // a LIVE HTMLCollection —
```

`querySelectorAll` returns a **static** `NodeList`: iterable with `for…of` / `forEach`, spread to an array for `map`. `getElementsBy*` return **live** collections — iterate them while removing elements and you skip every other one (the classic bug). Selectors are the CSS engine's: attribute selectors, `:not()`, `:nth-child`, `>` child and descendant combinators all work.

Reading and writing content

```
el.textContent;                  // all text of the subtree, no markup; setting it
el.innerText;                    // rendered text only (respects CSS display/visibi
el.innerHTML;                    // the markup; setting it PARSES the string as HTM
el.insertAdjacentHTML("beforeend", html);   // parse and insert without replacing
```

Rule: `textContent` **for data,** `innerHTML` **only for markup you built from trusted parts**. `el.innerHTML = "<b>" + userName + "</b>"` is an XSS vulnerability the moment `userName` contains `<img src=x onerror=…>`. If you must build markup from data, escape it (`& < > " '` → entities) or build elements with `createElement` and set `textContent`.

Attributes versus properties

```
<input id="q" value="initial" class="wide" data-user-id="42">
```

```
q.getAttribute("value");   // "initial" — the HTML attribute, what was in the mark
q.value;                   // the current value the user typed — a PROPERTY of the
q.value = "x";             // changes what is shown; the attribute stays "initial"
q.className;               // "wide"         q.classList.add("active"); q.classLis
q.dataset.userId;          // "42" — data-* attributes, kebab-case → camelCase, al
q.hidden = true;           // boolean property ⇔ the `hidden` attribute; el.disabl
q.style.backgroundColor = "red";   // inline style — prefer classes; getComputedSt
```

Attributes are strings in the markup; properties are the live object's state. For most attributes they reflect each other (`id`, `class` ↔ `className`, `href`, `disabled`), but `value` and `checked` split into "default" (attribute) and "current" (property) after user interaction — the source of "why does getAttribute return the old value". Use properties in code; use `setAttribute` for custom or ARIA attributes.

Creating, inserting, removing

```
const li = document.createElement("li");
li.textContent = item.name;
li.dataset.id = item.id;
li.append(icon, " ", label);                  // append: several nodes and strings;
list.prepend(li);  li.before(x);  li.after(y);  li.replaceWith(z);  li.remove();
list.insertBefore(li, list.firstChild);       // the older API
const clone = template.content.cloneNode(true);   // <template> + cloneNode for re

const frag = document.createDocumentFragment();   // build off-screen, insert once
for (const item of items) frag.append(renderItem(item));
list.replaceChildren(frag);                       // one DOM mutation instead of n
```

Inserting a node that is already in the document **moves** it. Removing a node leaves it in memory as long as you hold a reference (and its listeners with it).

Reflow, repaint, and batching

Changing geometry (size, position, font, adding elements) makes the browser recompute layout (**reflow**); changing colours only repaints; transforms and opacity can be composited without either. Reading a layout property (`offsetHeight`, `getBoundingClientRect()`, `getComputedStyle`) after a write forces a synchronous reflow so the read is accurate — a loop that writes then reads per item ("layout thrashing") reflows n times. Batch: read everything first, then write everything; build in a fragment and insert once; toggle a class instead of setting several styles; use `requestAnimationFrame` for visual updates so they land once per frame. Frameworks' virtual DOMs exist to do this batching for you.

Forms

`form.elements.name`, `input.value` (always a string — convert), `checkbox.checked`, `select.value` / `selectedOptions`, `input.valueAsNumber` / `valueAsDate`, `new FormData(form)` for all fields at once (`Object.fromEntries(new FormData(form))`), `form.requestSubmit()`, and the constraint API (`input.validity`, `required`, `pattern`, `setCustomValidity`) for validation the browser renders. Always validate again on the server.

Common mistakes

- `innerHTML` with user data (XSS); `innerText` in loops (forces layout).
- Iterating a live `getElementsByClassName` collection while removing elements.

Reading `getAttribute("value")` for the current input value; setting `style` in loops instead of a class.

One DOM insertion per item instead of a fragment; write-read-write-read layout thrashing.

Forgetting that `querySelector` returns `null` (then `.textContent` throws).

Holding references to removed nodes (and their listeners) forever.

Interview angle

"Attribute versus property?" Markup string versus live object state; `value` / `checked` diverge after user input.

"`textContent` versus `innerHTML` versus `innerText`?" Text (safe) versus parsed markup (XSS risk) versus rendered text (layout cost).

"Live versus static collections?" `getElementsBy*` update as the DOM changes; `querySelectorAll` is a snapshot.

"What is reflow and how do you minimise it?" Layout recomputation after geometry changes; batch writes, avoid interleaved reads, use fragments and classes.

"How do you safely render user input?" `textContent`, or escape entities before any `innerHTML`.

Key takeaways

The DOM is a tree of nodes; elements have attributes, children and properties; text is a child node.

`querySelector(All)` for selection (static), `closest` / `matches` for delegation; beware live collections.

`textContent` for data, `innerHTML` only for trusted markup; properties over attributes in code; `dataset` for data-*.

Create with `createElement` / `append`, batch with fragments and `replaceChildren`, read layout before writing.

Forms: `value` is a string, `FormData` for the whole form, browser validation plus server validation.

EXERCISE 1

A selector engine over a tree

The starter parses an indented outline (`tag#id.class.class text…`, two spaces per level) into a tree with `tag`, `id`, `classes`, `text`, `children` and `parent`. Implement `matches(node, simple)` for compound simple selectors (`li`, `#app`, `.item.active`, `li.item`), `querySelectorAll(scope, selector)` with the **descendant combinator** (space-separated parts, each matching a descendant of a previous match, no

duplicates, document order), `textContent(node)` (own text and descendants' text joined by single spaces), and `closest(node, selector)` (nearest ancestor-or-self). After `---`: `qsa <selector>` prints `qsa <selector>: <count> → <tag#id.classes, …>` (or `-`); `text <selector>` prints the JSON textContent of the first match (or `null`); `closest <start-selector> <selector>` prints the match or `null`.

Example (excerpt) for a shop page:

```
qsa li: 2 → li.item.active, li.item
qsa ul span: 1 → span.badge
qsa .missing: 0 → -
text ul: "Pen Ink new"
closest .badge .item: li.item
```

INPUT 1

```
div#app.main
  h1 Shop
  ul.list
    li.item.active Pen
    li.item Ink
      span.badge new
  p.note Free shipping
---
qsa li
qsa .item.active
qsa ul span
qsa #app p
qsa .missing
text ul
text li.active
closest .badge .item
closest .badge #app
```

OUTPUT 1

```
qsa li: 2 → li.item.active, li.item
qsa .item.active: 1 → li.item.active
qsa ul span: 1 → span.badge
qsa #app p: 1 → p.note
qsa .missing: 0 → -
text ul: "Pen Ink new"
text li.active: "Pen"
closest .badge .item: li.item
closest .badge #app: div#app.main
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
// Build a tree from indented lines: `tag#id.class.other text...`
function parseTree(treeLines) {
  const root = { tag: "document", id: "", classes: [], text: "", children: [], parent: nu
  const stack = [root];
  for (const raw of treeLines) {
    const depth = raw.match(/^ */)[0].length / 2;
    const [head, ...words] = raw.trim().split(/\s+/);
    const tag = head.match(/^[a-z0-9]+/)?.[0] ?? "div";
    const id = head.match(/#([\w-]+)/)?.[1] ?? "";
    const classes = [...head.matchAll(/\.([\w-]+)/g)].map((m) => m[1]);
    const node = { tag, id, classes, text: words.join(" "), children: [], parent: null, c
    while (stack.length > 1 && stack.at(-1).depth >= depth) stack.pop();
    node.parent = stack.at(-1);
    stack.at(-1).children.push(node);
    stack.push(node);
  }
  return root;
}
const describe = (n) => `${n.tag}${n.id ? "#" + n.id : ""}${n.classes.map((c) => "." + c)
const [treeText, commandText] = input.split("\n---\n");
const root = parseTree(treeText.split("\n").filter((l) => l.trim() !== ""));
function matches(node, simple) {
  const tag = simple.match(/^[a-z0-9]+/)?.[0];
  const id = simple.match(/#([\w-]+)/)?.[1];
  const classes = [...simple.matchAll(/\.([\w-]+)/g)].map((m) => m[1]);
  return (!tag || node.tag === tag) && (!id || node.id === id) && classes.every((c) => no
}
function* descendants(node) { for (const c of node.children) { yield c; yield* descendant
function querySelectorAll(scope, selector) {
  const parts = selector.trim().split(/\s+/);
  let current = [scope];
  for (const part of parts) {
    const next = new Set();
    for (const n of current) for (const d of descendants(n)) if (matches(d, part)) next.a
    current = [...next];
  }
  return current;
}
const querySelector = (scope, selector) => querySelectorAll(scope, selector)[0] ?? null;
function textContent(node) {
  return [node.text, ...node.children.map(textContent)].filter(Boolean).join(" ").trim();
}
function closest(node, selector) { for (let n = node; n && n.parent; n = n.parent) if (ma
for (const line of commandText.split("\n").filter((l) => l.trim() !== "")) {
  const [cmd, ...rest] = line.trim().split(/\s+/);
  if (cmd === "qsa") { const found = querySelectorAll(root, rest.join(" ")); console.log(
  else if (cmd === "text") { const n = querySelector(root, rest.join(" ")); console.log(`
  else if (cmd === "closest") { const start = querySelector(root, rest[0]); const found =
}
```

EXERCISE 2

Attribute, property and the dirty-value flag

Model an `<input>`: an `attributes` map plus live properties `value`, `checked`, `className`, `dataset`, and a `dirty` flag. Commands: `setAttr <name> <value>` sets the attribute — and, for `value`, updates the property **only while the user has not typed**; `type <text>` is a user edit (property only, sets `dirty`); `setProp <name> <value>` sets a property (`checked` as boolean; `value` also marks dirty); `addClass` / `removeClass` / `toggleClass` maintain `className` and the `class` attribute together; `data <kebab-name> <value>` sets `data-*` and the camelCase `dataset` key; `get` prints three lines: `value: attr=<JSON|null> prop=<JSON> dirty=<bool>`, `class: attr=<JSON|null> className=<JSON> list=<JSON>`, `dataset=<JSON> checked=<bool>`.

Example: `setAttr value initial`, `get`, `type hello`, `setAttr value later`, `get` →

```
value: attr="initial" prop="initial" dirty=false
…
value: attr="later" prop="hello" dirty=true
```

Once the user has typed, the attribute no longer drives the property — exactly the real DOM's behaviour and the reason `getAttribute("value")` is the wrong way to read an input.

INPUT 1

```
setAttr value initial
get
type hello
setAttr value later
get
setAttr class wide
addClass active
toggleClass active
addClass open
data user-id 42
setProp checked true
get
```

OUTPUT 1

```
value: attr="initial" prop="initial"
dirty=false
class: attr=null className="" list=[]
dataset={} checked=false
value: attr="later" prop="hello"
dirty=true
class: attr=null className="" list=[]
dataset={} checked=false
value: attr="later" prop="hello"
dirty=true
class: attr="wide open" className="wide
open" list=["wide","open"]
dataset={"userId":"42"} checked=true
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() ≡
// A model of an <input>: attributes (the markup) and properties (live state), with the d
const el = { attributes: new Map(), value: "", dirty: false, checked: false, className: "
const camel = (s) => s.replace(/-([a-z])/g, (_, c) => c.toUpperCase());
function setAttribute(name, value) {
  el.attributes.set(name, value);
  if (name ≡ "value" && !el.dirty) el.value = value;          // the attribute drives t
  if (name ≡ "class") el.className = value;
  if (name.startsWith("data-")) el.dataset[camel(name.slice(5))] = value;
}
const classList = {
  get list() { return el.className.split(/\s+/).filter(Boolean); },
  sync(list) { el.className = list.join(" "); el.attributes.set("class", el.className); }
  add(c) { const l = this.list; if (!l.includes(c)) l.push(c); this.sync(l); },
  remove(c) { this.sync(this.list.filter((x) => x ≢ c)); },
  toggle(c) { this.list.includes(c) ? this.remove(c) : this.add(c); },
};
for (const line of lines) {
  const [cmd, a, ...rest] = line.trim().split(/\s+/);
  const v = rest.join(" ");
  if (cmd ≡ "setAttr") setAttribute(a, v);
  else if (cmd ≡ "type") { el.value = a; el.dirty = true; }                  // a user
  else if (cmd ≡ "setProp") { el[a] = a ≡ "checked" ? v ≡ "true" : v; if (a ≡ "va
  else if (cmd ≡ "addClass") classList.add(a);
  else if (cmd ≡ "removeClass") classList.remove(a);
  else if (cmd ≡ "toggleClass") classList.toggle(a);
  else if (cmd ≡ "data") setAttribute(`data-${a}`, v);
  else if (cmd ≡ "get") {
    console.log(`value: attr=${JSON.stringify(el.attributes.get("value") ?? null)} prop=$
    console.log(`class: attr=${JSON.stringify(el.attributes.get("class") ?? null)} classN
    console.log(`dataset=${JSON.stringify(el.dataset)} checked=${el.checked}`);
  }
}
```

QUIZ

1. After the user types in an input, `input.getAttribute("value")` returns…

A. What they typed

B. The original markup value — the attribute is the default; the live text is the `value` **property**

C. `null`

D. An error

2. Rendering a user's comment with `el.innerHTML = comment`…

A. Is fine

B. Is an XSS vulnerability — the string is parsed as HTML; use `textContent` or escape

C. Is faster

D. Strips scripts automatically

3. Removing elements while looping over `document.getElementsByClassName("x")`…

A. Works

B. Skips elements — the collection is live and shrinks under the loop; use `querySelectorAll` (static) or iterate a copy

C. Throws

D. Is faster

4. Reading `el.offsetHeight` after changing a style, inside a loop over many elements…

A. Is free

B. Forces a synchronous reflow per iteration (layout thrashing) — read everything first, then write

C. Is cached

D. Only repaints

5. `data-user-id="42"` is read in JavaScript as…

A. `el.dataset["user-id"]`

B. `el.dataset.userId` — kebab-case maps to camelCase; the value is a string

C. `el.userId`

D. `el.getAttribute("userId")`

ANSWER KEY · THE DOM — THE TREE BEHIND THE PAGE, AND HOW TO CHANGE IT SAFELY

1. B `value` and `checked` diverge into default (attribute) and current (property) once the user interacts.

2. B `<img src=x onerror=…>` runs the moment it is inserted.

3. B `querySelectorAll` returns a snapshot.

4. B Fragments and `requestAnimationFrame` batch writes.

5. B `getAttribute("data-user-id")` also works.

LESSON 2 · 13 MIN

Events — propagation, delegation and the event object

Everything a user does reaches your code as an **event** dispatched on a DOM node and travelling through the tree — down from the document to the target (**capture**), then back up (**bubble**). Understanding that path explains delegation (one listener on a parent handling thousands of children), why `stopPropagation` is usually the wrong fix, what `target` and `currentTarget` mean, and how a framework's synthetic event system works. This lesson covers the listener API, the three phases, the event object and its defaults, delegation, custom events, and the listener-lifecycle mistakes that leak memory.

Listening

```
button.addEventListener("click", onClick);                        // any number of
button.addEventListener("click", onClick, { once: true });        // auto-removed a
window.addEventListener("scroll", onScroll, { passive: true });   // promise never
button.removeEventListener("click", onClick);                     // needs the SAME
const ac = new AbortController();
el.addEventListener("input", onInput, { signal: ac.signal });     // ac.abort() rem
el.onclick = fn;                                                  // the property f
```

Listeners run in registration order, synchronously, before `dispatchEvent` returns. An exception in one does not stop the others (it is reported as an uncaught error). `this` inside a non-arrow listener is `currentTarget`.

The three phases

Dispatching `click` on a `<button>` inside `<li>` inside `<ul>` inside `<body>`:

> **Capture**: `window → document → html → body → ul → li` — listeners registered with `{ capture: true }` fire, outermost first.
>
> **Target**: listeners on `button` fire (both capture and bubble ones, in registration order).
>
> **Bubble**: `li → ul → body → html → document → window` — ordinary listeners fire, innermost first.

Most events bubble (`click`, `input`, `keydown`, `submit`); a few do not (`focus` / `blur` — use `focusin` / `focusout`; `load`, `scroll` on elements, `mouseenter` / `mouseleave`). `event.eventPhase` says which phase is running; `event.composedPath()` lists the route (including shadow DOM boundaries).

The event object

```
el.addEventListener("click", (e) => {
  e.type;            // "click"
  e.target;          // the element the event happened ON (deepest) — a <span> ins
  e.currentTarget;   // the element this listener is attached to — what you usuall
  e.preventDefault();      // cancel the browser's default action: following a lin
  e.stopPropagation();     // no further nodes in the path receive it (listeners o
  e.stopImmediatePropagation();   // not even the remaining listeners on this node
  e.defaultPrevented; e.bubbles; e.cancelable; e.timeStamp; e.isTrusted;   // isTr
  // per-type fields: e.key / e.code (keyboard), e.clientX / e.button (pointer), e
});
```

`preventDefault` and `stopPropagation` are independent: cancelling a form's submit does not stop the event bubbling, and stopping the bubble does not cancel the submit. Passive listeners cannot `preventDefault` (the call is ignored with a warning) — that promise is what lets the browser scroll before your handler runs.

Delegation

```
list.addEventListener("click", (e) => {
  const item = e.target.closest("[data-id]");        // walk up from the deepest e
  if (!item || !list.contains(item)) return;         // clicked between items, or
  const action = e.target.closest("[data-action]")?.dataset.action;
  if (action === "remove") remove(item.dataset.id);
});
```

One listener on the container handles every current *and future* child: no per-item wiring, no leaks when items are removed, one function to maintain. `e.target` is wherever the click landed (an icon inside a button inside the item), so `closest` climbs to the element you care about. Every framework's event system is delegation at the root plus a lookup table.

`stopPropagation` — mostly a smell

Stopping propagation to keep an outer handler from running couples the two handlers invisibly: the outer one (analytics, a "click outside to close" handler, a delegated listener) silently never sees the event. Prefer letting events bubble and having the outer handler check `e.target` (`if (dialog.contains(e.target)) return`). Legitimate uses are rare: nested interactive components with genuinely conflicting semantics.

Custom events

```
const ev = new CustomEvent("cart:add", { detail: { id, qty }, bubbles: true, cance
const notCancelled = item.dispatchEvent(ev);     // false if a listener called pre
document.addEventListener("cart:add", (e) ⇒ e.detail.id);
```

Custom events decouple components: the item announces, anyone listens, nobody imports anybody. `bubbles: true` if a distant ancestor should hear it; `detail` carries the payload. In Node the same shape is `EventEmitter` / `EventTarget` (module 9).

Input, change, keyboard, pointer

`input` fires on every keystroke/paste (value changing); `change` on commit (blur for text, immediately for checkboxes/selects). `keydown` (repeats while held, cancelable — the one for shortcuts; use `e.key` like `"Enter"` / `"a"`, `e.code` for physical position) versus `keyup`; `keypress` is deprecated. Pointer events (`pointerdown/move/up`) unify mouse, touch and pen; `click` still fires for taps and for keyboard activation of buttons (accessibility for free — do not replace buttons with divs). `submit` fires on the **form**, not the button; handle it there and call `e.preventDefault()` for JavaScript submission.

Lifecycle and leaks

A listener keeps its closure alive, and the closure keeps whatever it references. Adding a listener on `window` / `document` from a component and never removing it leaks the component; adding a listener per render leaks n copies. Remove in teardown (frameworks' unmount hooks), or use `{ signal }` with one `AbortController` per component, or delegate to a parent that lives as long as the children. Listeners on a removed element go away with it — if nothing else references the element.

Common mistakes

- `removeEventListener` with a fresh arrow (nothing removed).
- Using `target` where `currentTarget` was meant (the click landed on a child).
- `stopPropagation` to silence an outer handler; `preventDefault` on a passive listener.
- Listening for `submit` on the button, `change` when `input` was needed, `keypress` at all.
- Adding listeners in a loop over items instead of delegating; forgetting teardown.
- Replacing `<button>` with a `<div onclick>` and losing keyboard/AT accessibility.

Interview angle

"Explain event propagation." Capture down from `window` to the target, target phase, bubble back up; `capture: true` listens on the way down.

"`target` versus `currentTarget`?" Where the event originated versus where the listener is attached.

"What is event delegation and why use it?" One listener on an ancestor handling descendants via `closest`; fewer listeners, works for future children, no per-item cleanup.

"`preventDefault` versus `stopPropagation`?" Cancel the browser's default action versus stop the event travelling further; independent.

"What does `{ passive: true }` do?" Declares the listener will not `preventDefault`, letting the browser scroll immediately.

Key takeaways

`addEventListener(type, fn, { once, passive, capture, signal })`; remove with the same reference or a signal; listeners run synchronously in order.

Capture (down) → target → bubble (up); most events bubble; `focus` / `blur` do not (`focusin` / `focusout` do).

`target` is where it happened, `currentTarget` where you listened; `preventDefault` and `stopPropagation` are independent.

Delegate with one ancestor listener + `closest`; avoid `stopPropagation` as coupling.

`CustomEvent` with `detail` for component communication; `submit` on the form; `input` vs `change`; `keydown` + `e.key`.

EXERCISE 1

A three-phase event dispatcher

Lines containing `>` declare node paths (`document > body > ul#list > li#a > button#b`; a node's name is its id, or its tag when it has none). `on <node> <type> <capture|bubble> <label> [stop|stopImmediate|prevent]` registers a listener. `dispatch <node> <type>` runs the three phases: **capture** from the root down to the target's parent (capture listeners only), **target** (capture-registered listeners first, then bubble-registered — the modern spec), **bubble** from the parent up to the root (bubble listeners). Each invoked listener prints `<label> @<currentTarget> target=<target> phase=<capture|target|bubble>`; `stop` stops further **nodes**, `stopImmediate` also the remaining listeners on the same node, `prevent` sets the flag. After dispatch print `dispatch <type> on <node>: defaultPrevented=<bool>`.

Example (excerpt): listeners on document (capture), list (bubble and capture), a (bubble), b (bubble with prevent, capture) →

```
doc-capture @document target=b phase=capture
list-capture @list target=b phase=capture
b-capture @b target=b phase=target
b-bubble @b target=b phase=target
a-bubble @a target=b phase=bubble
list-bubble @list target=b phase=bubble
dispatch click on b: defaultPrevented=true
```

INPUT 1

```
document > body > ul#list > li#a >
button#b
on document click capture doc-capture
on list click bubble list-bubble
on list click capture list-capture
on a click bubble a-bubble
on b click bubble b-bubble prevent
on b click capture b-capture
dispatch b click
```

OUTPUT 1

```
doc-capture @document target=b
phase=capture
list-capture @list target=b phase=capture
b-capture @b target=b phase=target
b-bubble @b target=b phase=target
a-bubble @a target=b phase=bubble
list-bubble @list target=b phase=bubble
dispatch click on b: defaultPrevented=true
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =
// Nodes are declared as paths: `document > body > ul#list > li#a > button#b`
const nodes = new Map();      // id → { id, parent, listeners: [] }
function declare(pathLine) {
  const ids = pathLine.split(">").map((s) => s.trim().replace(/^[a-z0-9]*#?/, "") || s.tr
  let parent = null;
  for (const id of ids) {
    if (!nodes.has(id)) nodes.set(id, { id, parent, listeners: [] });
    parent = nodes.get(id);
  }
}
function dispatch(targetId, type) {
  const target = nodes.get(targetId);
  const path = [];
  for (let n = target.parent; n; n = n.parent) path.unshift(n);      // root ... parent
  const event = { type, target: targetId, defaultPrevented: false, stopped: false, stoppe
  const run = (node, phase, filter) => {
    for (const l of node.listeners) {
      if (l.type !== type || !filter(l)) continue;
      console.log(`${l.label} @${node.id} target=${event.target} phase=${phase}`);
      if (l.flag === "prevent") event.defaultPrevented = true;
      if (l.flag === "stop") event.stopped = true;
      if (l.flag === "stopImmediate") { event.stopped = true; event.stoppedImmediate = tr
    }
  };
  for (const n of path) { if (event.stopped) break; run(n, "capture", (l) => l.capture);
  if (!event.stopped) { run(target, "target", (l) => l.capture); if (!event.stoppedImmedi
  for (const n of [...path].reverse()) { if (event.stopped) break; run(n, "bubble", (l) =
  console.log(`dispatch ${type} on ${targetId}: defaultPrevented=${event.defaultPrevented
}
for (const line of lines) {
  const text = line.trim();
  if (text.includes(">")) { declare(text); continue; }
  const [cmd, node, type, phase, label, flag] = text.split(/\s+/);
  if (cmd === "on") nodes.get(node).listeners.push({ type, capture: phase === "capture",
  else if (cmd === "dispatch") dispatch(node, type);
}
```

EXERCISE 2

Delegation with closest

A container holds items (`add <id> [action]` , `remove <id>`). Register **one** delegated click listener on the container and one `once` listener that prints `first click on the container!` . A click arrives as a target path such as `item3.label` (a child inside item 3), `item2` , or `gap` (between items); the delegated listener resolves the item the way `e.target.closest("[data-id]")` would (the first segment), prints `click <path>: <action or select> on <id>` or `click <path>: ignored (no item)` , and counts handled actions. Finish with `listenersAdded=<n> active=<n> items=<n> handled=<JSON sorted by action>` .

Example: `add item1 open`, `add item2`, `click item1.icon`, `click item2`, `click gap`, `add item3 delete`, `click item3.label`, `remove item1`, `click item1` →

```
click item1.icon: open on item1
first click on the container!
click item2: select on item2
click gap: ignored (no item)
click item3.label: delete on item3
click item1: ignored (no item)
listenersAdded=2 active=1 items=2 handled={"delete":1,"open":1,"select":1}
```

Items added after the listener was registered still work; the removed one is simply not found.

INPUT 1

```
add item1 open
add item2
click item1.icon
click item2
click gap
add item3 delete
click item3.label
remove item1
click item1
```

OUTPUT 1

```
click item1.icon: open on item1
first click on the container!
click item2: select on item2
click gap: ignored (no item)
click item3.label: delete on item3
click item1: ignored (no item)
listenersAdded=2 active=1 items=2 handled=
{"delete":1,"open":1,"select":1}
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =
// The container holds items; a click lands on a path like `item3.icon` (the icon span ir
const items = new Map();   // id → { id, action }
const handled = new Map();
let listenerCount = 0;
const listeners = [];
const addListener = (fn, opts = {}) => { listeners.push({ fn, ...opts }); listenerCount+
const closestItem = (targetPath) => { const id = targetPath.split(".")[0]; return items.g
addListener((targetPath) => {
  const item = closestItem(targetPath);
  if (!item) { console.log(`click ${targetPath}: ignored (no item)`); return; }
  const action = item.action ?? "select";
  handled.set(action, (handled.get(action) ?? 0) + 1);
  console.log(`click ${targetPath}: ${action} on ${item.id}`);
});
addListener(() => console.log("first click on the container!"), { once: true });
function dispatchClick(targetPath) {
  for (const l of [...listeners]) {
    l.fn(targetPath);
    if (l.once) listeners.splice(listeners.indexOf(l), 1);
  }
}
for (const line of lines) {
  const [cmd, a, b] = line.trim().split(/\s+/);
  if (cmd === "add") items.set(a, { id: a, action: b });
  else if (cmd === "remove") items.delete(a);
  else if (cmd === "click") dispatchClick(a);
}
console.log(`listenersAdded=${listenerCount} active=${listeners.length} items=${items.siz
```

QUIZ

1. A click on a `<span>` inside a `<button>` with a listener on the button: `e.target` and `e.currentTarget` are…

A. Both the button

B. The span and the button, respectively — where it happened versus where you listened

C. Both the span

D. The document and the button

2. Event delegation means…

A. Each item gets its own listener

B. One listener on an ancestor handles events from all descendants, present and future, via `e.target.closest(...)`

C. Using `stopPropagation`

D. Custom events

3. `e.preventDefault()` on a form's `submit` …

A. Stops the event bubbling

B. Cancels the browser's submission; the event still propagates — the two are independent

C. Removes the listener

D. Resets the form

4. `el.removeEventListener("click", () ⇒ handle())` after `el.addEventListener("click", () ⇒ handle())` …

A. Removes the listener

B. Removes nothing — a fresh arrow is a different function; keep the reference or use an `AbortSignal`

C. Throws

D. Removes all listeners

5. `{ passive: true }` on a scroll listener…

A. Makes it run later

B. Promises the handler will not call `preventDefault`, so the browser can scroll immediately

C. Removes it after one call

D. Runs it in capture

ANSWER KEY · EVENTS — PROPAGATION, DELEGATION AND THE EVENT OBJECT

1. B Use `currentTarget` (or `closest`) for the element you care about.

2. B Fewer listeners, no per-item cleanup, works for dynamically added children.

3. B `stopPropagation` does not cancel the default action either.

4. B `{ signal }` lets one `abort()` remove every listener of a component.

5. B `once: true` is the auto-remove option.

LESSON 3 · 13 MIN

fetch, URLs, CORS and talking to servers

`fetch` is how a page (and, since Node 18, a server) talks HTTP: a promise of a `Response`, from which you read a body once. It looks simple and hides four decisions that every client gets wrong at least once — a 404 is **not** a rejection, a body can be read only once, requests to other origins are governed by CORS (which the browser enforces and the server configures), and nothing times out unless you make it. This lesson covers the request and response objects, error handling done right, JSON in and out, `URL` / `URLSearchParams` for building addresses, `AbortController`, CORS as a mental model, credentials and cookies, and the tools beyond `fetch` — SSE and WebSockets.

The shape

```
const res = await fetch("/api/users?limit=10", {
  method: "POST",                                    // GET by default
  headers: { "Content-Type": "application/json", Accept: "application/json" },
  body: JSON.stringify({ name }),                    // string | FormData | UR
  signal: controller.signal,                         // cancellation / timeout
  credentials: "same-origin",                        // "include" to send cook
});
res.ok;            // status 200-299
res.status;        // 404
res.headers.get("content-type");
const data = await res.json();      // parses the body — ONCE; also .text(), .blob
```

`fetch` **rejects only on network failure** (DNS, connection refused, CORS blocked, aborted). A 404 or 500 resolves normally with `ok: false`. Every client needs the check:

```
async function getJson(url, init) {
  const res = await fetch(url, init);
  if (!res.ok) {
    const detail = await res.text().catch(() => "");                    // read
    throw new HttpError(res.status, `${init?.method ?? "GET"} ${url} → ${res.statu
  }
  return res.json();                                                    // may
}
```

The body is a stream consumed once: `await res.json()` after `await res.text()` throws `body used already`. `res.clone()` before reading if two consumers need it.

Building URLs

```
const url = new URL("/api/search", location.origin);            // absolute from a
url.searchParams.set("q", userText);                             // encoded correct
url.searchParams.append("tag", "x"); url.searchParams.append("tag", "y");   // rep
url.pathname = `/api/users/${encodeURIComponent(id)}`;           // path segments n
String(url);                                                      // "https://…/api
Object.fromEntries(new URL(location.href).searchParams);          // read the curre
```

Never concatenate query strings by hand; `URLSearchParams` handles encoding, repeats and ordering. Distinguish `encodeURIComponent` (a single value — encodes `/ ? & =`) from `encodeURI` (a whole URL — leaves them). Module 9 covered the same classes in Node.

Timeouts, cancellation, retries

```
const controller = new AbortController();
const timer = setTimeout(() => controller.abort(), 8000);
try {
  return await getJson(url, { signal: controller.signal });
} catch (err) {
  if (err.name === "AbortError") throw new Error(`timed out: ${url}`);
  throw err;
} finally {
  clearTimeout(timer);
}
```

`fetch` never times out by itself. Abort also when the user navigates away or types a new search (cancel the previous request so a slow old response cannot overwrite a fast new one — the *race* bug). Retry only idempotent requests (GET, PUT, DELETE with care) on network errors and 5xx/429, with backoff and jitter (module 8), honouring `Retry-After`. A 4xx is the client's fault; retrying it repeats the mistake.

CORS, as a mental model

The **same-origin policy**: a page at `https://app.example` may read responses only from `https://app.example` (scheme + host + port). To read from `https://api.example`, the **server** must opt in with `Access-Control-Allow-Origin: https://app.example` (or `*` — never with credentials). **Simple** requests (GET/HEAD/POST with form-ish content types, no custom headers) are sent directly and the browser checks the response header. Anything else — `Content-Type: application/json`, an `Authorization` header, PUT/DELETE — triggers a **preflight** `OPTIONS`

request first; the server answers with `Access-Control-Allow-Methods/Headers` (and `Max-Age` to cache it). Cookies cross-origin need `credentials: "include"` on the client and `Access-Control-Allow-Credentials: true` plus an explicit origin on the server.

Consequences: CORS is enforced by the **browser**, not by curl or Node; a "CORS error" means the *server* did not allow it (or preflight failed); the request may have reached the server even though your code cannot read the response; and CORS protects users from malicious pages, not APIs from clients — authentication does that.

Sending data

JSON body with `Content-Type: application/json` for APIs; `FormData` (multipart, no content-type header — the browser sets the boundary) for files; `URLSearchParams` as a body for form-encoded endpoints. Responses: check `content-type` before `json()` when a server may return HTML error pages; treat a `204` as having no body. Compression, caching (`Cache-Control`, `ETag` / `If-None-Match`) and content negotiation are headers you set once and forget.

Cookies and tokens

Session cookies set by the server with `HttpOnly` cannot be read by JavaScript (good: XSS cannot steal them) and are sent automatically same-origin; `SameSite=Lax` (default) blocks most cross-site sending — the CSRF defence. Bearer tokens in an `Authorization` header are sent by your code, work cross-origin, and must be stored somewhere JavaScript can read (memory is safest; `localStorage` is readable by any XSS). The module on storage goes further.

Beyond request/response

Server-Sent Events (`new EventSource(url)`): server → client stream of text events over one HTTP connection, auto-reconnecting; simple and firewall-friendly for feeds, progress, live updates.

WebSocket (`new WebSocket(url)`): bidirectional, low-latency messages; needs its own reconnection and heartbeats; for chat, games, collaborative editing.

Streaming `fetch`: `res.body.getReader()` reads chunks as they arrive — progress bars, incremental rendering.

`navigator.sendBeacon`: fire-and-forget analytics that survives page unload.

`XMLHttpRequest` is the pre-2015 API — upload progress events are its one remaining advantage.

Common mistakes

Treating a resolved `fetch` as success; not reading the error body.

Reading the body twice; `res.json()` on HTML.

Hand-built query strings; `encodeURI` for a value.

No timeout; not aborting superseded requests (stale response wins the race).

Retrying POSTs or 4xx; expecting CORS to be fixable from the client; `*` with credentials.

Tokens in `localStorage` without weighing XSS; `credentials: "include"` when same-origin was enough.

Interview angle

"Does `fetch` reject on a 404?" No — only on network failure or abort; check `res.ok`.

"What is CORS?" Browser-enforced relaxation of the same-origin policy: the server declares which origins may read its responses; non-simple requests preflight with OPTIONS.

"How do you time out a fetch?" `AbortController` + `setTimeout(abort)`, clear in `finally`; also abort superseded requests.

"Why can't `res.json()` be called twice?" The body is a one-shot stream; `clone()` first if needed.

"Where do you store an auth token?" HttpOnly cookie with SameSite (not readable by JS) or in memory; `localStorage` is exposed to XSS.

Key takeaways

`fetch` resolves for any HTTP status; check `res.ok`, read the error body, throw a typed error; parse the body once.

Build URLs with `URL` / `URLSearchParams`; `encodeURIComponent` for values in paths.

Timeouts and cancellation via `AbortController`; retry only idempotent requests with backoff; abort stale requests.

CORS is the server allowing origins, enforced by the browser; preflights for non-simple requests; credentials need explicit origins.

SSE for server push, WebSocket for bidirectional, streaming fetch for incremental bodies.

EXERCISE 1

A fetch client that handles the real cases

The starter defines a fake `fetch` over a route table (a 200 with JSON, a 404 with a JSON error body, a 500 with an HTML body, a slow route, and an unknown route that rejects with `TypeError("Failed to fetch")` like a network failure); its Response's body can be read **once**. Implement `fakeFetch` as described (resolve after the route's delay; reject with an `AbortError`-named error when the signal aborts) and `getJson(url, timeoutMs)`: abort via `AbortController` after the timeout (clear the timer in `finally`), throw `HttpError(status, "GET <url> → <status>", detail)` with the error body as `detail` when `!res.ok`, else return `res.json()`. Line 1 is the timeout; each following line a URL. Print `<url>: ok <JSON>`, `<url>: HttpError <status> detail=<JSON>`, `<url>: timed out after <ms> ms` or `<url>: <name>: <message>`; finally read a body twice and print `second read: TypeError: body used already`.

Example: `30` then `/api/user`, `/api/missing`, `/api/broken`, `/api/slow`, `/api/nowhere` →

```
/api/user: ok {"id":1,"name":"Ada"}
/api/missing: HttpError 404 detail="{\"error\":\"no such user\"}"
/api/broken: HttpError 500 detail="<html>Internal Server Error</html>"
/api/slow: timed out after 30 ms
/api/nowhere: TypeError: Failed to fetch
second read: TypeError: body used already
```

INPUT 1

```
30
/api/user
/api/missing
/api/broken
/api/slow
/api/nowhere
```

OUTPUT 1

```
/api/user: ok {"id":1,"name":"Ada"}
/api/missing: HttpError 404 detail="
{\"error\":\"no such user\"}"
/api/broken: HttpError 500 detail="
<html>Internal Server Error</html>"
/api/slow: timed out after 30 ms
/api/nowhere: TypeError: Failed to fetch
second read: TypeError: body used already
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() ≡
const sleep = (ms, v) => new Promise((r) => setTimeout(r, ms, v));
const routes = {
  "/api/user": { status: 200, delay: 5, body: { id: 1, name: "Ada" } },
  "/api/missing": { status: 404, delay: 5, body: { error: "no such user" } },
  "/api/broken": { status: 500, delay: 5, text: "<html>Internal Server Error</html>" },
  "/api/slow": { status: 100, delay: 100, body: { ok: true } },
};
routes["/api/slow"].status = 200;
function fakeFetch(url, { signal } = {}) {
  const route = routes[new URL(url, "http://localhost").pathname];
  return new Promise((resolve, reject) => {
    if (!route) return reject(new TypeError("Failed to fetch"));            // network-le
    const timer = setTimeout(() => {
      let used = false;
      const text = route.text ?? JSON.stringify(route.body);
      const read = () => { if (used) return Promise.reject(new TypeError("body used alrea
      resolve({ ok: route.status ⩾ 200 && route.status < 300, status: route.status, text
    }, route.delay);
    signal?.addEventListener("abort", () => { clearTimeout(timer); reject(Object.assign(n
  });
}
class HttpError extends Error { constructor(status, message, detail) { super(message); th
async function getJson(url, timeoutMs) {
  const controller = new AbortController();
  const timer = setTimeout(() => controller.abort(), timeoutMs);
  try {
    const res = await fakeFetch(url, { signal: controller.signal });
    if (!res.ok) {
      const detail = await res.text().catch(() => "");
      throw new HttpError(res.status, `GET ${url} → ${res.status}`, detail);
    }
    return await res.json();
  } finally {
    clearTimeout(timer);
  }
}
async function main() {
  const timeoutMs = Number(lines[0]);
  for (const url of lines.slice(1).map((l) => l.trim())) {
    try {
      console.log(`${url}: ok ${JSON.stringify(await getJson(url, timeoutMs))}`);
    } catch (err) {
      if (err instanceof HttpError) console.log(`${url}: HttpError ${err.status} detail=$
      else if (err.name === "AbortError") console.log(`${url}: timed out after ${timeoutM
      else console.log(`${url}: ${err.name}: ${err.message}`);
    }
  }
  const res = await fakeFetch("/api/user");
  await res.text();
  await res.json().then(() => console.log("second read ok"), (e) => console.log(`second r
```

```
}
main();
```

EXERCISE 2

Building URLs the right way

Drive the real `URL` class with commands: `base <url>`, `resolve <relative>` (relative to the current URL), `set <key> <value…>`, `append <key> <value…>`, `delete <key>`, `path <segments…>` (each segment `encodeURIComponent`-ed), `hash [value]` (no value clears it), `show` (prints `String(url)`), `parts` (prints `origin= pathname= search= hash= host= port=<JSON>`), `params` (prints the entries as JSON).

Example: `base https://shop.example.com/catalog/items?page=1#top`, `set q blue pens & more`, `append tag office`, `append tag sale`, `show`, `resolve ../about`, `show`, `path api users 42/edit`, `delete tag`, `hash`, `show` →

```
https://shop.example.com/catalog/items?page=1&q=blue+pens+%26+more&tag=office&tag=sale#to
https://shop.example.com/about
https://shop.example.com/api/users/42%2Fedit
```

`URLSearchParams` encoded the ampersand; `encodeURIComponent` kept the slash inside the segment as `%2F`.

INPUT 1

```
base
https://shop.example.com/catalog/items?
page=1#top
parts
set q blue pens & more
append tag office
append tag sale
show
params
resolve ../about
show
path api users 42/edit
delete tag
hash
show
```

OUTPUT 1

```
origin=https://shop.example.com
pathname=/catalog/items search=?page=1 hash=#top
host=shop.example.com port=""
https://shop.example.com/catalog/items?
page=1&q=blue+pens+%26+more&tag=office&tag=sale#top
params=[["page","1"],["q","blue pens & more"],
["tag","office"],["tag","sale"]]
https://shop.example.com/about
https://shop.example.com/api/users/42%2Fedit
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =
let url = null;
for (const line of lines) {
  const [cmd, ...args] = line.trim().split(/\s+/);
  if (cmd === "base") url = new URL(args[0]);
  else if (cmd === "resolve") url = new URL(args[0], url);
  else if (cmd === "set") url.searchParams.set(args[0], args.slice(1).join(" "));
  else if (cmd === "append") url.searchParams.append(args[0], args.slice(1).join(" "));
  else if (cmd === "delete") url.searchParams.delete(args[0]);
  else if (cmd === "path") url.pathname = "/" + args.map(encodeURIComponent).join("/");
  else if (cmd === "hash") url.hash = args[0] ?? "";           // no argument clears it
  else if (cmd === "show") console.log(String(url));
  else if (cmd === "parts") console.log(`origin=${url.origin} pathname=${url.pathname} se
  else if (cmd === "params") console.log(`params=${JSON.stringify([...url.searchParams])}
}
```

QUIZ

1. `await fetch("/missing")` for a URL that returns 404…

A. Rejects

B. Resolves with `res.ok === false` — fetch rejects only on network failure or abort; check `ok` yourself

C. Returns `null`

D. Retries

2. Calling `res.json()` after `res.text()` on the same Response…

A. Works

B. Throws `body used already` — the body is a one-shot stream; `clone()` first if two readers are needed

C. Returns the cached text

D. Returns `undefined`

3. A "CORS error" in the console means…

A. Your JavaScript has a bug

B. The browser refused to let the page read the response because the **server** did not allow the page's origin — fix it on the server

C. The server is down

D. The URL is wrong

4. To time out a fetch after 5 s you…

A. Pass `{ timeout: 5000 }`

B. Create an `AbortController`, `setTimeout(() ⇒ controller.abort(), 5000)`, pass `signal`, and clear the timer in `finally`

C. Use `Promise.race` with `setTimeout` only

D. Cannot

5. `encodeURIComponent` versus `encodeURI`:

A. Synonyms

B. `encodeURIComponent` encodes a single value (including `/ ? & =`); `encodeURI` leaves URL structure intact — for whole URLs

C. `encodeURI` is stricter

D. Neither is needed with `URL`

ANSWER KEY · FETCH, URLS, CORS AND TALKING TO SERVERS

1. B Read the error body for detail and throw a typed error.

2. B Decide the format once, or clone.

3. B curl and Node do not enforce CORS; the request may even have reached the server.

4. B Abort also cancels superseded requests so a slow old response cannot overwrite a new one.

5. B `URLSearchParams` handles query values; `encodeURIComponent` for path segments.

LESSON 4 · 12 MIN

Storage, cookies, history and the browser APIs worth knowing by name

A page has several places to keep state between reloads and several ways to observe the world around it, and each has a shape that decides what it is good for: `localStorage` is a small synchronous string dictionary; cookies travel to the server with every request and carry security attributes; IndexedDB is a real asynchronous database; the History API lets a single-page app own the URL. Around them sit the observer APIs that replaced polling and scroll handlers, and workers that give the page real threads. This lesson is a working map — what each API is, its one rule, and which task it fits — with the security flags that matter on cookies.

Web Storage: `localStorage` and `sessionStorage`

```
localStorage.setItem("theme", "dark");            // strings only — objects become
localStorage.setItem("prefs", JSON.stringify(prefs));
const prefs = JSON.parse(localStorage.getItem("prefs") ?? "{}");   // getItem → nu
localStorage.removeItem("theme"); localStorage.clear(); localStorage.length; local
window.addEventListener("storage", (e) ⇒ { e.key; e.oldValue; e.newValue; });   /
```

Per **origin**, synchronous, ~5 MB, survives reloads and restarts (`localStorage`) or lives for one tab (`sessionStorage`). Rules: values are strings (serialise), reads block the main thread (fine for small values, not for a loop), it is readable by any script on the origin (never store secrets or tokens you would not hand to an XSS), it can be disabled or full (`setItem` throws `QuotaExceededError` — wrap in `try`), and the `storage` event is a free cross-tab channel. Good for: preferences, drafts, a small cache, the theme. Not for: large data, sensitive data, anything a server must trust.

Cookies

```
Set-Cookie: session=abc123; Path=/; Max-Age=86400; Secure; HttpOnly; SameSite=Lax
```

Cookies are set by the server (or by `document.cookie = "k=v; …"` for non-HttpOnly ones) and sent automatically with matching requests — that automatic sending is both their purpose (sessions) and their risk (CSRF). The attributes:

- `Secure` — only over HTTPS.
- `HttpOnly` — invisible to `document.cookie` ; the session cookie is unreadable by injected scripts. Always for session ids.

`SameSite` — `Strict` (never sent on cross-site requests), `Lax` (sent on top-level navigations with safe methods, not on cross-site POSTs or subresources — the default in modern browsers), `None` (sent everywhere; requires `Secure`). `Lax` / `Strict` is the modern CSRF defence; anti-CSRF tokens remain for `None`.

`Path`, `Domain` — which URLs and hosts receive it; a `Domain=example.com` cookie reaches subdomains.

`Max-Age` / `Expires` — otherwise a session cookie deleted when the browser closes. ~4 KB each.

Reading `document.cookie` gives `"a=1; b=2"` — parse it; writing sets one cookie at a time. Cookies go with every request to the origin, so they cost bandwidth; keep them small (an id, not the data).

IndexedDB, and the Cache API

IndexedDB is an asynchronous, transactional, object-store database per origin with indexes and hundreds of megabytes of room — for offline data, large caches, files. Its raw API is verbose and event-based; use a wrapper (`idb`) that promisifies it. **Cache API** (`caches.open`) stores request/response pairs and is the store service workers use for offline assets. Rule of thumb: small config → `localStorage`; structured or large data → IndexedDB; HTTP responses for offline → Cache API; everything server-side → the server.

URL, location and history

```
location.href; location.pathname; location.search; location.hash; location.origin;
location.assign(url); location.replace(url);            // navigate (replace: no h
history.pushState({ page: 2 }, "", "/items?page=2");    // change the URL WITHOUT
history.replaceState(state, "", url);                  // rewrite the current entr
history.back(); history.forward(); history.go(-2);
window.addEventListener("popstate", (e) => render(location.pathname, e.state));
window.addEventListener("hashchange", …);              // for #-based routing
```

A client-side router is: intercept link clicks (same-origin, no modifier keys) → `pushState` → render; listen to `popstate` → render. The server must serve the app shell for every route (or the deep link 404s on refresh). `state` is stored with the entry (structured-clone-able, small). Scroll restoration and focus management are the router's job too.

Observers

`IntersectionObserver` — "is this element visible / near the viewport?" without scroll handlers: lazy-load images, infinite scroll, analytics impressions. Callback with entries and

`isIntersecting`; thresholds; `rootMargin` to prefetch early.

`ResizeObserver` — element size changes (not just window resize): responsive components, canvas sizing.

`MutationObserver` — DOM changes (children, attributes, text) — for integrating with code that mutates the DOM outside your control; batched, asynchronous.

`PerformanceObserver` — long tasks, layout shifts, paint timings (Core Web Vitals).

Observers are asynchronous and batched; they replace polling and per-event handlers with something the browser can schedule efficiently.

Timing and frames

`requestAnimationFrame(cb)` runs before the next paint — for visual updates and measuring; `requestIdleCallback` when the browser is idle — for non-urgent work; `setTimeout` / `setInterval` as in Node but clamped in background tabs; `performance.now()` monotonic time; `performance.mark/measure` for custom timing; `document.visibilityState` + `visibilitychange` to pause work in hidden tabs.

Workers and the rest

Web Workers run scripts on another thread with message passing (`postMessage` / `onmessage`, structured clone or transfer) — for CPU-heavy work (parsing, image processing, crypto) so the UI stays responsive; no DOM access. **Service Workers** are a network proxy for the origin: offline caching, push notifications, background sync — the base of PWAs. Others to recognise: `navigator.clipboard` (async, permission-gated), `Notification` (permission), `Geolocation`, `matchMedia` (media queries in JS), `Blob` / `File` / `FileReader` / `URL.createObjectURL`, `structuredClone`, `BroadcastChannel` (cross-tab messaging without storage), `Intl` (module 10), Canvas/WebGL/WebAudio, `crypto.subtle`. Each is permission- and feature-detected: `if ("clipboard" in navigator)`.

Common mistakes

Objects into `localStorage` without `JSON.stringify`; no `try` around `setItem`; secrets in storage.

Session cookies without `HttpOnly` / `Secure` / `SameSite`; `SameSite=None` without `Secure`.

Expecting `popstate` after your own `pushState`; SPA routes that 404 on refresh because the server does not serve the shell.

Scroll handlers for visibility instead of `IntersectionObserver`; `setInterval` polling for DOM changes instead of `MutationObserver`.

Heavy computation on the main thread when a worker was available.

Interview angle

"`localStorage` versus cookies?" Client-only string store, not sent to the server, readable by any script; cookies travel with requests, carry security attributes, are the session mechanism.

"What do `HttpOnly`, `Secure` and `SameSite` do?" Hide from JavaScript; HTTPS only; restrict cross-site sending (CSRF defence).

"How does a single-page router change the URL?" `history.pushState` + render; `popstate` for back/forward; server serves the shell for all routes.

"How would you lazy-load images?" `IntersectionObserver` with a `rootMargin`, or `loading="lazy"`.

"When do you use a Web Worker?" CPU-bound work that would block the UI thread; communicate by messages.

Key takeaways

Web Storage: per-origin, synchronous, strings, ~5 MB, readable by XSS; `storage` event across tabs; `try` around writes.

Cookies: sent automatically; `Secure` + `HttpOnly` + `SameSite=Lax/Strict` for sessions; small.

IndexedDB for structured/large data (use a wrapper); Cache API for offline responses.

`pushState` / `popstate` are the SPA router; the server must serve the shell everywhere.

Observers (Intersection/Resize/Mutation/Performance) over polling; `requestAnimationFrame` for visuals; workers for CPU work.

EXERCISE 1

A Web Storage model with quota and cross-tab events

Implement `WebStorage(quotaBytes, name, bus)`: `setItem(key, value)` coerces the value with `String()`, throws an error named `QuotaExceededError` (`quota of <n> bytes exceeded`) when the total of key and value lengths would exceed the quota, stores, and emits a `storage` event that **other** tabs print as `[<tab>] storage event: key=<k> old=<JSON> new=<JSON>`; `getItem` returns `null` when missing; `removeItem` emits with `new=null`; `key(i)`, `length`. Two tabs `A` and `B` share the origin. Commands: `<tab> set <k> <v…>`, `<tab> get <k>`, `<tab> remove <k>`, `<tab> json <k> <json>` (round-trips through

`JSON.stringify` / `parse`), `<tab> raw <k>` (tries to store the object `{ a: 1 }` directly), `<tab> keys`. Print each command's outcome or `<tab> <cmd> <k>: <error name>: <message>`.

Example (excerpt), quota 40: `A set theme dark` → the event line in tab B, then `A set theme`; `A raw obj` → the stored text would be `[object Object]` (16 chars) and exceeds the quota; `B remove theme` → B's own storage is empty, so `A get theme` still gives `"dark"`.

```
  [B] storage event: key=theme old=null new="dark"
A set theme
B get theme → null
```

INPUT 1

```
40
A set theme dark
B get theme
A json prefs {"n":1,"list":[1,2]}
A raw obj
A set big
01234567890123456789012345678901234567890123456789
A keys
B remove theme
A get theme
```

OUTPUT 1

```
  [B] storage event: key=theme old=null
new="dark"
A set theme
B get theme → null
  [B] storage event: key=prefs old=null
new="{\"n\":1,\"list\":[1,2]}"
A json prefs → {"n":1,"list":[1,2]}
A raw obj: QuotaExceededError: quota of 40
bytes exceeded
A set big: QuotaExceededError: quota of 40
bytes exceeded
A keys=["theme","prefs"] length=2
B removed theme
A get theme → "dark"
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =
class WebStorage {
  #data = new Map();
  constructor(quotaBytes, name, bus) { this.quota = quotaBytes; this.name = name; this.bu
  #bytes() { let n = 0; for (const [k, v] of this.#data) n += k.length + v.length; returr
  setItem(key, value) {
    const text = String(value);                                   // objects become "[
    const old = this.#data.get(key) ?? null;
    const next = this.#bytes() - (old ? key.length + old.length : 0) + key.length + text.
    if (next > this.quota) throw Object.assign(new Error(`quota of ${this.quota} bytes ex
    this.#data.set(key, text);
    this.bus.emit({ key, oldValue: old, newValue: text, source: this.name });
  }
  getItem(key) { return this.#data.get(key) ?? null; }
  removeItem(key) { if (this.#data.has(key)) { const old = this.#data.get(key); this.#dat
  key(i) { return [...this.#data.keys()][i] ?? null; }
  get length() { return this.#data.size; }
}
const bus = { tabs: [], emit(e) { for (const t of this.tabs) if (t.name !== e.source) cor
const quota = Number(lines[0]);
const A = new WebStorage(quota, "A", bus), B = new WebStorage(quota, "B", bus);
bus.tabs.push(A, B);
const tabs = { A, B };
for (const line of lines.slice(1)) {
  const [tab, cmd, key, ...rest] = line.trim().split(/\s+/);
  const s = tabs[tab];
  try {
    if (cmd === "set") { s.setItem(key, rest.join(" ")); console.log(`${tab} set ${key}`)
    else if (cmd === "get") console.log(`${tab} get ${key} → ${JSON.stringify(s.getItem(
    else if (cmd === "remove") { s.removeItem(key); console.log(`${tab} removed ${key}`);
    else if (cmd === "json") { s.setItem(key, JSON.stringify(JSON.parse(rest.join(" "))))
    else if (cmd === "raw") { s.setItem(key, { a: 1 }); console.log(`${tab} raw ${key} →
    else if (cmd === "keys") console.log(`${tab} keys=${JSON.stringify(Array.from({ lengt
  } catch (err) {
    console.log(`${tab} ${cmd} ${key}: ${err.name}: ${err.message}`);
  }
}
```

EXERCISE 2

A cookie jar that applies the flags

Parse `Set-Cookie:` lines into `{ name, value, path (default /), secure, httpOnly, sameSite (default Lax), maxAge }`. For each `request <url> <same-site|cross-site> <method> <navigation|subresource>` decide per cookie whether it is sent, with the first failing rule as the reason: `expired` (Max-Age ≤ 0), `insecure` (Secure on a non-https URL), `path` (request path does not start with the cookie path), `samesite-none-requires-secure`, `samesite-strict` (cross-site), `samesite-lax` (cross-site unless a top-level GET navigation). Print `<METHOD> <url> [<site>, <kind>] → Cookie:`

`<name=value; …> | blocked: <name(reason), …>` (`(none)` when nothing is sent; omit the blocked part when empty). Finish with `readableByScript=<names without HttpOnly>`.

Example (excerpt) for a session (Secure, HttpOnly, Lax), a Strict theme, a None-without-Secure tracker, an expired promo and a Secure cart on `/shop`:

```
POST https://shop.example/shop/checkout [cross-site, subresource] → Cookie: (none) | blo
GET https://shop.example/help [cross-site, navigation] → Cookie: session=abc | blocked:
readableByScript=theme,tracker,promo,cart
```

INPUT 1

```
Set-Cookie: session=abc; Path=/; Secure;
HttpOnly; SameSite=Lax
Set-Cookie: theme=dark; Path=/;
SameSite=Strict
Set-Cookie: tracker=t1; Path=/;
SameSite=None
Set-Cookie: promo=x; Path=/shop; Secure;
SameSite=Lax; Max-Age=0
Set-Cookie: cart=3; Path=/shop; Secure;
SameSite=Lax
request https://shop.example/shop/cart
same-site GET navigation
request http://shop.example/shop/cart
same-site GET navigation
request https://shop.example/shop/checkout
cross-site POST subresource
request https://shop.example/help cross-
site GET navigation
```

OUTPUT 1

```
GET https://shop.example/shop/cart [same-
site, navigation] → Cookie: session=abc;
theme=dark; cart=3 | blocked:
tracker(samesite-none-requires-secure),
promo(expired)
GET http://shop.example/shop/cart [same-
site, navigation] → Cookie: theme=dark |
blocked: session(insecure),
tracker(samesite-none-requires-secure),
promo(expired), cart(insecure)
POST https://shop.example/shop/checkout
[cross-site, subresource] → Cookie:
(none) | blocked: session(samesite-lax),
theme(samesite-strict), tracker(samesite-
none-requires-secure), promo(expired),
cart(samesite-lax)
GET https://shop.example/help [cross-site,
navigation] → Cookie: session=abc |
blocked: theme(samesite-strict),
tracker(samesite-none-requires-secure),
promo(expired), cart(path)
readableByScript=theme,tracker,promo,cart
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =
// `Set-Cookie: name=value; Path=/; Secure; HttpOnly; SameSite=Lax; Max-Age=3600` lines,
function parseSetCookie(header) {
  const [pair, ...attrs] = header.replace(/^Set-Cookie:\s*/i, "").split(";").map((s) => s
  const [name, value = ""] = pair.split("=");
  const cookie = { name, value, path: "/", secure: false, httpOnly: false, sameSite: "Lax
  for (const attr of attrs) {
    const [k, v = ""] = attr.split("=");
    const key = k.toLowerCase();
    if (key === "path") cookie.path = v;
    else if (key === "secure") cookie.secure = true;
    else if (key === "httponly") cookie.httpOnly = true;
    else if (key === "samesite") cookie.sameSite = v[0].toUpperCase() + v.slice(1).toLowe
    else if (key === "max-age") cookie.maxAge = Number(v);
  }
  return cookie;
}
function shouldSend(cookie, req) {
  if (cookie.maxAge !== null && cookie.maxAge <= 0) return "expired";
  if (cookie.secure && req.url.protocol !== "https:") return "insecure";
  if (!req.url.pathname.startsWith(cookie.path)) return "path";
  if (cookie.sameSite === "None" && !cookie.secure) return "samesite-none-requires-secure
  if (req.site === "cross-site") {
    if (cookie.sameSite === "Strict") return "samesite-strict";
    if (cookie.sameSite === "Lax" && !(req.kind === "navigation" && req.method === "GET")
  }
  return null;
}
const jar = [];
for (const line of lines) {
  const text = line.trim();
  if (/^set-cookie:/i.test(text)) { jar.push(parseSetCookie(text)); continue; }
  if (!text.startsWith("request ")) continue;
  const [, urlText, site, method, kind] = text.split(/\s+/);
  const req = { url: new URL(urlText), site, method, kind };
  const sent = [], blocked = [];
  for (const c of jar) { const why = shouldSend(c, req); (why ? blocked : sent).push(why
  console.log(`${method} ${urlText} [${site}, ${kind}] → Cookie: ${sent.join("; ") || "(
}
console.log(`readableByScript=${jar.filter((c) => !c.httpOnly).map((c) => c.name).join(",
```

QUIZ

1. `localStorage.setItem("user", { name: "Ada" })` stores…

A. The object

B. The string `"[object Object]"` — values are strings; `JSON.stringify` first

C. `null`

D. An error

2. The `storage` event fires…

A. In the tab that wrote

B. In **other** tabs of the same origin — a free cross-tab channel

C. On the server

D. Never

3. A session cookie should carry…

A. Nothing special

B. `Secure` (HTTPS only), `HttpOnly` (invisible to scripts) and `SameSite=Lax` or `Strict` (CSRF defence)

C. `SameSite=None` only

D. A long `Max-Age`

4. `history.pushState(state, "", "/items")` …

A. Loads `/items`

B. Changes the URL and adds a history entry without loading; your code renders — and `popstate` does **not** fire for it

C. Fires `popstate`

D. Reloads

5. Lazy-loading images as they approach the viewport is best done with…

A. A scroll handler measuring `getBoundingClientRect`

B. `IntersectionObserver` (with a `rootMargin`) or `loading="lazy"`

C. `setInterval`

D. `MutationObserver`

ANSWER KEY · STORAGE, COOKIES, HISTORY AND THE BROWSER APIS WORTH KNOWING BY NAME

1. B And `getItem` returns `null` for a missing key, so `JSON.parse(x ?? "{}")`.

2. B `BroadcastChannel` is the dedicated alternative.

3. B `SameSite=None` requires `Secure` and reopens CSRF unless tokens are used.

4. B `popstate` fires on back/forward; the server must serve the shell for deep links.

5. B Observers are batched and cheap; scroll handlers with layout reads thrash.

LESSON 5 · 13 MIN

Rendering performance and front-end security — the two things a page must not get wrong

A page can be correct and still fail its users in two ways: it can be **slow** — janky scrolling, a spinner over a white screen, an input that lags behind the keyboard — or it can be **unsafe** — a comment field that runs someone else's script in your users' sessions. Both have well-understood causes and defences. This lesson covers how the browser turns DOM and CSS into pixels and what makes that expensive; the loading path from HTML to interactive; the measurement tools and the vitals; then the front-end threat model: XSS and its defences (escaping, `textContent`, sanitisers, CSP), CSRF and `SameSite`, clickjacking, open redirects, third-party scripts and dependency risk.

From DOM to pixels

The browser builds the DOM and CSSOM, combines them into a **render tree**, computes **layout** (geometry: where everything is and how big), **paints** (draws pixels into layers), then **composites** the layers onto the screen. JavaScript can dirty any stage: changing text or a class → style recalculation → layout → paint → composite; changing only `transform` / `opacity` on a promoted layer → composite only (cheap — this is why animations use those two properties). Layout is proportional to the affected subtree; a change to `font-size` on `body` relays out everything. `requestAnimationFrame` batches your visual writes to just before the next frame; the frame budget at 60 Hz is **16.7 ms** for everything — your JavaScript, style, layout, paint. A 30 ms handler drops frames.

Layout thrashing (module lesson 1): interleaved reads (`offsetHeight`, `getBoundingClientRect`) and writes force synchronous layout per iteration. Read first, then write; or use `IntersectionObserver` / `ResizeObserver` instead of measuring in scroll handlers.

The loading path

HTML streams in and parses; a `<script>` without `defer` / `async` **blocks parsing** until it downloads and runs (put scripts at the end or use `defer`, which runs in order after parsing; `async` runs as soon as downloaded, unordered — for independent scripts like analytics); CSS in `<head>` blocks rendering until loaded (critical CSS inline, the rest deferred); fonts can block text (`font-display: swap`); images without dimensions cause layout shifts (set `width` / `height` or `aspect-ratio`). Code splitting (dynamic `import()` per route), lazy-loading images (`loading="lazy"`), preloading the critical request (`<link rel="preload">`), compression and caching headers, and shipping less JavaScript are the levers — the last one dwarfs the rest.

Measuring

Core Web Vitals name what users feel: **LCP** (Largest Contentful Paint — when the main content shows, target < 2.5 s), **INP** (Interaction to Next Paint — input responsiveness, < 200 ms), **CLS** (Cumulative Layout Shift — things jumping, < 0.1). DevTools' Performance panel shows the frame timeline (long tasks > 50 ms block input); Lighthouse audits the loading path; `PerformanceObserver` reports vitals from real users (RUM). Measure before optimising; the profile usually points at one or two culprits — a large bundle, a synchronous layout loop, an unbatched render.

Rendering strategies

Manual DOM updates get slow to *write correctly* long before they get slow to *run*; a **virtual DOM** (React) or **fine-grained reactivity** (Solid, Vue, Svelte) exists to keep UI = f(state) while batching writes and updating only what changed. A virtual DOM diff compares two trees and emits patches (props changed, children inserted/removed/moved — keys make moves cheap). Server rendering (SSR) and static generation send HTML first so LCP does not wait for JavaScript; hydration then attaches behaviour. The trade-offs are the frameworks' whole subject; the principle is constant: batch, diff, and touch the real DOM as little as possible.

XSS — the front-end vulnerability

Cross-site scripting: attacker-controlled text is rendered as **markup**, so their `<script>` or `onerror=` runs with your users' cookies, storage and API access. Sources: `innerHTML` / `outerHTML` / `insertAdjacentHTML` / `document.write` with data, `javascript:` and `data:` URLs in `href` / `src`, `eval` / `new Function` on data, unsanitised markdown, server templates without auto-escaping. Defences, in layers:

- **Render text as text**: `textContent`, `createElement` + properties, framework bindings (React escapes `{value}`; `dangerouslySetInnerHTML` is the escape hatch with the warning in its name).
- **Escape** at the point of insertion when markup is unavoidable: `& < > " '` → entities — context-aware (attribute, URL and script contexts differ).
- **Sanitise** untrusted HTML that must stay HTML (user-authored rich text) with an allow-list library (DOMPurify) — never a hand-written regex.
- **Content Security Policy** header: `script-src 'self'` (no inline scripts, no `eval`, only your origin) turns most XSS into a blocked-script console message; nonces/hashes for necessary inline scripts; `object-src 'none'`; report violations.

HttpOnly session cookies so a successful XSS cannot exfiltrate the session; tokens in memory rather than storage.

CSRF, clickjacking, redirects

CSRF: a malicious page makes the user's browser send a request to your site, with the user's cookies attached automatically. Defences: `SameSite=Lax/Strict` cookies (blocks the cross-site POST), anti-CSRF tokens in forms/headers for legacy cases, checking `Origin` / `Sec-Fetch-Site` headers, and never performing state changes on GET. **Clickjacking**: your page framed invisibly over a decoy so clicks land on your buttons — `Content-Security-Policy: frame-ancestors 'none'` (or `X-Frame-Options`). **Open redirects**: `?next=https://evil` after login — allow only relative paths or an allow-list. `target="_blank"` links get `rel="noopener"` by default now; add `noreferrer` when the destination should not learn the referrer.

Third parties and dependencies

Every `<script src="https://cdn…">` runs with full page privileges; a compromised CDN or tag manager is a total compromise. Pin with Subresource Integrity (`integrity="sha384-…"`), self-host where possible, limit with CSP, and audit the npm tree (module 9) — a front-end bundle is as much other people's code as a server. Store nothing in the page a script should not read.

Common mistakes

- Animating `top` / `left` / `width` instead of `transform` ; measuring layout inside scroll handlers.
- Blocking `<script>` tags in `<head>` ; images without dimensions (CLS); one giant bundle.
- `innerHTML` with any user-influenced string; sanitising with a regex; `href` from data without a scheme allow-list.
- No CSP, or a CSP with `'unsafe-inline'` that defends nothing; session cookies without `HttpOnly` / `SameSite` .
- State-changing GET endpoints; `next=` redirects without validation; third-party scripts without SRI.

Interview angle

"What happens between a DOM change and pixels?" Style → layout → paint → composite; `transform` / `opacity` skip layout and paint.

"`defer` versus `async`?" Both download in parallel; `defer` runs in order after parsing, `async` as soon as it arrives, unordered.

"What is XSS and how do you prevent it?" Untrusted text rendered as markup; render as text, escape, sanitise with an allow-list, CSP, HttpOnly cookies.

"What is CSRF and the modern defence?" Cross-site requests riding the user's cookies; `SameSite` cookies plus tokens where needed; no state changes on GET.

"Name the Core Web Vitals." LCP (loading), INP (responsiveness), CLS (visual stability).

Key takeaways

Style → layout → paint → composite; batch writes, avoid read/write interleaving, animate `transform` / `opacity`, stay inside the 16.7 ms frame.

Loading: `defer` scripts, critical CSS, image dimensions, code splitting, less JavaScript; measure LCP/INP/CLS.

XSS: text as text, escape, sanitise with an allow-list, CSP without `'unsafe-inline'`, HttpOnly cookies.

CSRF: `SameSite` cookies and tokens, no GET mutations; `frame-ancestors` against clickjacking; validate redirects.

Third-party scripts are full-privilege code: SRI, self-host, CSP, audit.

EXERCISE 1

Count the reflows

The starter models the layout engine: `write()` marks layout dirty, `read()` forces a reflow when dirty (counting it). Starting dirty each time, implement three strategies over the input heights and end each with the frame's own `read()`: **naive** — for each item write then read (measure after mutating); **batched** — read every item first, then write every item; **requestAnimationFrame** — read each item immediately but queue the writes in a fake `requestAnimationFrame`, then run the frame. Print `naive: reflows=<n> for <n> items`, `batched: reflows=<n>`, `requestAnimationFrame: reflows=<n>`, `sameResult=<all three produced the same doubled heights> out=<JSON>`.

Example: `10 20 30 40 50` →

```
naive: reflows=5 for 5 items
batched: reflows=2
requestAnimationFrame: reflows=2
sameResult=true out=[20,40,60,80,100]
```

Interleaving a read after every write costs one layout per item; separating the phases costs one layout per phase.

INPUT 1

```
10 20 30 40 50
```

OUTPUT 1

```
naive: reflows=5 for 5 items
batched: reflows=2
requestAnimationFrame: reflows=2
sameResult=true out=[20,40,60,80,100]
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
// A model of the layout engine: writes dirty the layout; a read while dirty forces a syn
const engine = { dirty: false, reflows: 0, write() { this.dirty = true; }, read() { if (t
const heights = lines[0].trim().split(/\s+/).map(Number);
function naive() {
  engine.reflows = 0; engine.dirty = true;
  const out = [];
  for (const h of heights) { engine.write(); engine.read(); out.push(h * 2); }     // wri
  engine.read();                                                                    // t
  return { reflows: engine.reflows, out };
}
function batched() {
  engine.reflows = 0; engine.dirty = true;
  const measured = heights.map((h) => { engine.read(); return h; });                // d
  const out = measured.map((h) => { engine.write(); return h * 2; });               // t
  engine.read();                                                                    // d
  return { reflows: engine.reflows, out };
}
function withFrame() {
  engine.reflows = 0; engine.dirty = true;
  const pending = [];
  const requestAnimationFrame = (cb) => pending.push(cb);                            //
  const out = [];
  for (const h of heights) { engine.read(); requestAnimationFrame(() => { engine.write();
  for (const cb of pending) cb();                                                   // t
  engine.read();
  return { reflows: engine.reflows, out };
}
const a = naive(), b = batched(), c = withFrame();
console.log(`naive: reflows=${a.reflows} for ${heights.length} items`);
console.log(`batched: reflows=${b.reflows}`);
console.log(`requestAnimationFrame: reflows=${c.reflows}`);
console.log(`sameResult=${JSON.stringify(a.out) === JSON.stringify(b.out) && JSON.stringi
```

EXERCISE 2

Sanitise untrusted HTML and check a CSP

Line 1 is untrusted HTML, line 2 a Content-Security-Policy header, the rest are script sources to test. Write `sanitize(markup)` : split into tags and text; keep only `b i p a br` (an `<a>` keeps only an `http(s)` `href` , re-emitted as `href="…" rel="noopener"`); drop every other tag — and the whole content of

`<script>` / `<style>` — recording what was dropped (`onclick`-style attributes as `onclick`, a rejected link as `a[href=javascript:]`); escape all text with an escaper that leaves existing entities alone. Print `sanitized=<html>` and `dropped=<comma list or →>`. Then `cspAllows(policy, source)` for the `script-src` directive: `inline` needs `'unsafe-inline'`, `eval` needs `'unsafe-eval'`, `self` needs `'self'`, a URL matches a host source exactly or a `*.` wildcard; no directive → nothing allowed. Print `<source>: allowed|blocked`.

Example (excerpt):

```
sanitized=<p>Hi <b>Ada</b> &amp;amp; <a href="https://x.io" rel="noopener">link</a><a>bad</a>
dropped=onclick,script,img,a[href=javascript:]
inline: blocked
https://a.trusted.net/x.js: allowed
https://evil.example/x.js: blocked
```

INPUT 1

```
<p>Hi <b>Ada</b> &amp;amp; <a
href="https://x.io"
onclick="steal()">link</a>
<script>alert(1)</script><img src=x
onerror="boom()"><a
href="javascript:evil()">bad</a></p>
script-src 'self'
https://cdn.example.com *.trusted.net;
object-src 'none'
self
inline
eval
https://cdn.example.com/lib.js
https://a.trusted.net/x.js
https://evil.example/x.js
```

OUTPUT 1

```
sanitized=<p>Hi <b>Ada</b> &amp;amp; <a
href="https://x.io" rel="noopener">link</a>
<a>bad</a></p>
dropped=onclick,script,img,a[href=javascript:]
self: allowed
inline: blocked
eval: blocked
https://cdn.example.com/lib.js: allowed
https://a.trusted.net/x.js: allowed
https://evil.example/x.js: blocked
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
const [html, cspLine, ...scripts] = input.split("\n").filter((l) => l.trim() !== "");
const ALLOWED_TAGS = new Set(["b", "i", "p", "a", "br"]);
const escapeHtml = (s) => s.replace(/&(?!#?\w+;)|[<>"']/g, (c) => ({ "&": "&amp;", "<": '
function sanitize(markup) {
  const tokens = markup.split(/(<[^>]+>)/);                                    // tags ar
  const dropped = [];
  let skipping = null;                                                         // inside <
  const out = tokens.map((tok) => {
    const m = /^<(\/?)([a-zA-Z0-9]+)([^>]*)>$/.exec(tok);
    if (!m) return skipping ? "" : escapeHtml(tok);
    const [, close, rawTag, attrText] = m;
    const tag = rawTag.toLowerCase();
    if (skipping) { if (close && tag === skipping) skipping = null; return ""; }
    if (!ALLOWED_TAGS.has(tag)) { dropped.push(tag); if (!close && (tag === "script" || t
    if (close) return `</${tag}>`;
    let attrs = "";
    if (tag === "a") {
      const href = /href\s*=\s*"([^"]*)"/i.exec(attrText)?.[1];
      if (href && /^https?:\/\//i.test(href)) attrs = ` href="${escapeHtml(href)}" rel="r
      else if (href) dropped.push(`a[href=${href.split(":")[0]}:]`);
    }
    for (const on of attrText.matchAll(/\s(on\w+)=/gi)) dropped.push(on[1].toLowerCase())
    return `<${tag}${attrs}>`;
  });
  return { html: out.join(""), dropped };
}
function cspAllows(policy, source) {
  const directive = policy.split(";").map((d) => d.trim()).find((d) => d.startsWith("scri
  const sources = directive.split(/\s+/).slice(1);
  if (source === "inline") return sources.includes("'unsafe-inline'");
  if (source === "eval") return sources.includes("'unsafe-eval'");
  if (source === "self") return sources.includes("'self'");
  const host = new URL(source).host;
  return sources.some((s) => {
    if (s.startsWith("'")) return false;
    const pattern = s.replace(/^https?:\/\//, "");
    return pattern.startsWith("*.") ? host.endsWith(pattern.slice(1)) : host === pattern;
  });
}
const { html: clean, dropped } = sanitize(html);
console.log(`sanitized=${clean}`);
console.log(`dropped=${dropped.join(",") || "-"}`);
for (const s of scripts.map((l) => l.trim())) console.log(`${s}: ${cspAllows(cspLine, s)
```

QUIZ

1. Animating `transform` and `opacity` is cheap because…

A. They are shorter to type

B. They can be handled by the compositor without layout or paint; `top` / `left` / `width` trigger layout

C. They run on the GPU always

D. They are cached

2. `<script defer>` versus `<script async>`:

A. Identical

B. Both download in parallel; `defer` runs in order after parsing, `async` runs as soon as it arrives, in any order

C. `async` blocks parsing

D. `defer` runs first

3. A Content-Security-Policy of `script-src 'self'` …

A. Blocks all scripts

B. Allows scripts only from the page's origin — inline scripts and `eval` are blocked, which neutralises most XSS payloads

C. Allows CDNs

D. Only affects images

4. CSRF works because…

A. Scripts can read cookies

B. The browser attaches the user's cookies to requests a malicious page triggers — `SameSite` cookies and tokens stop it; never mutate on GET

C. Passwords are weak

D. CORS is misconfigured

5. The three Core Web Vitals are…

A. TTFB, FCP, TTI

B. LCP (loading), INP (responsiveness), CLS (visual stability)

C. CPU, memory, network

D. HTML, CSS, JS size

ANSWER KEY · RENDERING PERFORMANCE AND FRONT-END SECURITY — THE TWO THINGS A PAGE MUST NOT GET WRONG

1. B Style → layout → paint → composite; skipping the first two keeps the 16.7 ms frame.

2. B A plain `<script>` blocks parsing until it runs.

3. B Adding `'unsafe-inline'` defeats it; use nonces or hashes for necessary inline scripts.

4. B XSS steals from within the page; CSRF rides from outside it.

5. B Measure real users with `PerformanceObserver`; audit with Lighthouse.

CHECKPOINT · 24 MIN

Checkpoint — The DOM and the browser

This checkpoint covers the DOM tree and selection, attributes versus properties, `textContent` versus `innerHTML`, batching and reflow; event propagation, `target` / `currentTarget`, delegation, `preventDefault` / `stopPropagation`, custom events; `fetch` semantics, `URL` / `URLSearchParams`, timeouts with `AbortController`, CORS; Web Storage, cookies and their attributes, the History API, observers and workers; the rendering pipeline, loading path, vitals, and the XSS/CSRF defences.

How it works. Twelve questions and three programs; 70% on the questions and every program accepted clears the module. The programs model browser mechanisms in Node — a DOM tree, an event dispatcher, a router — because the judge has no browser; the ideas are exactly the browser's.

Before you start, make sure you can answer:

- Attribute versus property for `value`; which content API is safe for user data; why `getElementsByClassName` bites when removing.
- The three phases; what `closest` does in a delegated handler; why `stopPropagation` is usually a smell; `passive`.
- Why a 404 does not reject; how to time out a fetch; what a CORS error means and who fixes it.
- The three cookie flags and what each defends; `pushState` versus `popstate`; when to use IndexedDB.
- Which CSS properties skip layout; `defer` versus `async`; the layered XSS defences; the CSRF defence.

The programs are a virtual-DOM differ that emits a minimal patch list, a client-side router with parameterised routes and a history stack, and a stale-while-revalidate fetch cache modelled with timers.

EXERCISE 1

A virtual-DOM differ

Two lines hold JSON trees: `{ tag, props, children }` where a child is a tree or a string. Write `diff(a, b, path, patches)` producing, in document order: `REPLACE <path>` when a text node and an element swap, `REPLACE <path> <oldTag> → <newTag>` for different tags (do not descend), `TEXT <path> <oldJSON> → <newJSON>`, `PROPS <path> <+k=v ~k=v -k …>` (added in new-key order, then changed, then removed), and for children by index `INSERT <path> <"text" or <tag>>` / `REMOVE <path>`; paths are `root`, `root/0`, `root/1/0`. Print the patches one per line or `no changes`.

Example: a list whose class and role change, whose first item loses an id, whose second item's text changes, and which gains a fourth item →

```
PROPS root ~class="list active" +role="list"
PROPS root/0 -id
TEXT root/1/0 "Ink" → "Blue Ink"
INSERT root/3 <li>
```

INPUT 1

```
{"tag":"ul","props":
{"class":"list"},"children":
[{"tag":"li","props":
{"id":"a"},"children":["Pen"]},
{"tag":"li","props":{},"children":
["Ink"]},{"tag":"li","props":
{},"children":["Pad"]}]}
{"tag":"ul","props":{"class":"list
active","role":"list"},"children":
[{"tag":"li","props":{},"children":
["Pen"]},{"tag":"li","props":
{},"children":["Blue Ink"]},
{"tag":"li","props":{},"children":
["Pad"]},{"tag":"li","props":
{},"children":["Clip"]}]}
```

OUTPUT 1

```
PROPS root ~class="list active"
+role="list"
PROPS root/0 -id
TEXT root/1/0 "Ink" → "Blue Ink"
INSERT root/3 <li>
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =
const [oldTree, newTree] = lines.map((l) => JSON.parse(l));
function diff(a, b, path, patches) {
  if (typeof a === "string" || typeof b === "string") {
    if (typeof a !== typeof b) patches.push(`REPLACE ${path}`);
    else if (a !== b) patches.push(`TEXT ${path} ${JSON.stringify(a)} → ${JSON.stringify
    return;
  }
  if (a.tag !== b.tag) { patches.push(`REPLACE ${path} <${a.tag}> → <${b.tag}>`); return
  const ap = a.props ?? {}, bp = b.props ?? {};
  const changes = [];
  for (const k of Object.keys(bp)) { if (!(k in ap)) changes.push(`+${k}=${JSON.stringify
  for (const k of Object.keys(ap)) if (!(k in bp)) changes.push(`-${k}`);
  if (changes.length) patches.push(`PROPS ${path} ${changes.join(" ")}`);
  const ac = a.children ?? [], bc = b.children ?? [];
  const n = Math.max(ac.length, bc.length);
  for (let i = 0; i < n; i++) {
    const childPath = `${path}/${i}`;
    if (i >= ac.length) patches.push(`INSERT ${childPath} ${typeof bc[i] === "string" ? J
    else if (i >= bc.length) patches.push(`REMOVE ${childPath}`);
    else diff(ac[i], bc[i], childPath, patches);
  }
}
const patches = [];
diff(oldTree, newTree, "root", patches);
console.log(patches.length ? patches.join("\n") : "no changes");
```

EXERCISE 2

A client-side router with a history stack

Lines `route <pattern>` declare routes with `:param` segments and a trailing `*` (captured as `rest`); compile each to a regex. Keep a history array starting at `["/"]` with an index. Commands: `push <path?query>` (discard the forward stack, append, move) prints `pushState <path> → <match>`; `replace <path>` prints `replaceState …`; `back` / `forward` move the index and print `popstate <path> → <match>` (or `back: nothing to go back to` / `forward: nothing ahead`); `stack` prints `stack=<JSON> index=<n>`. A match is `matched <pattern> params=<JSON> query=<JSON of the query string>` (first declared route wins) or `404 <pathname>`.

Example (excerpt):

```
pushState /users/42?tab=posts → matched /users/:id params={"id":"42"} query={"tab":"post
pushState /files/docs/a.pdf → matched /files/* params={"rest":"docs/a.pdf"} query={}
popstate /users/42/posts/7 → matched /users/:id/posts/:postId params={"id":"42","postId"
pushState /nowhere → 404 /nowhere
```

A `push` after going back drops the entries ahead — exactly what the browser does.

INPUT 1

```
route /
route /users/:id
route /users/:id/posts/:postId
route /files/*
push /users/42?tab=posts
push /users/42/posts/7
push /files/docs/a.pdf
back
back
stack
push /nowhere
forward
back
replace /users/1
stack
```

OUTPUT 1

```
pushState /users/42?tab=posts → matched /users/:id
params={"id":"42"} query={"tab":"posts"}
pushState /users/42/posts/7 → matched
/users/:id/posts/:postId params=
{"id":"42","postId":"7"} query={}
pushState /files/docs/a.pdf → matched /files/*
params={"rest":"docs/a.pdf"} query={}
popstate /users/42/posts/7 → matched
/users/:id/posts/:postId params=
{"id":"42","postId":"7"} query={}
popstate /users/42?tab=posts → matched /users/:id
params={"id":"42"} query={"tab":"posts"}
stack=["/","/users/42?
tab=posts","/users/42/posts/7","/files/docs/a.pdf"]
index=1
pushState /nowhere → 404 /nowhere
forward: nothing ahead
popstate /users/42?tab=posts → matched /users/:id
params={"id":"42"} query={"tab":"posts"}
replaceState /users/1 → matched /users/:id params=
{"id":"1"} query={}
stack=["/","/users/1","/nowhere"] index=1
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
const routeLines = lines.filter((l) => l.startsWith("route "));
const commands = lines.filter((l) => !l.startsWith("route "));
const routes = routeLines.map((l) => {
  const pattern = l.split(/\s+/)[1];
  const keys = [];
  const source = pattern.replace(/\//g, "\\/").replace(/:(\w+)/g, (_, k) => { keys.push(k
  return { pattern, keys, re: new RegExp(`^${source}$`) };
});
function match(pathWithQuery) {
  const url = new URL(pathWithQuery, "http://app.local");
  const query = Object.fromEntries(url.searchParams);
  for (const r of routes) {
    const m = r.re.exec(url.pathname);
    if (m) return `matched ${r.pattern} params=${JSON.stringify(Object.fromEntries(r.keys
  }
  return `404 ${url.pathname}`;
}
const history = ["/"];
let index = 0;
const render = (why) => console.log(`${why} ${history[index]} → ${match(history[index])}
for (const line of commands) {
  const [cmd, arg] = line.trim().split(/\s+/);
  if (cmd === "push") { history.splice(index + 1); history.push(arg); index++; render("pu
  else if (cmd === "replace") { history[index] = arg; render("replaceState"); }
  else if (cmd === "back") { if (index > 0) { index--; render("popstate"); } else console
  else if (cmd === "forward") { if (index < history.length - 1) { index++; render("popsta
  else if (cmd === "stack") console.log(`stack=${JSON.stringify(history)} index=${index}`
}
```

EXERCISE 3

A stale-while-revalidate cache

`fetchValue(key)` (in the starter) takes 10 ms, counts calls and returns `<key>#<count>`. Implement `SwrCache(ttlMs).get(key)`: on a **miss** fetch, store with a timestamp and return `{ status: "miss", value }`; when the entry is younger than the TTL return `{ status: "hit", value }`; when it is **stale** return the stale value immediately as `{ status: "stale", value }` and start **one** background revalidation that replaces the entry when it completes (a second stale read while it is in flight must not start another). Line 1 is the TTL; commands `get <key>` print `get <key>: <status> <value> fetches=<count>` and `wait <ms>` prints `(waited <ms> ms)`.

Example: TTL `40`, then `get a`, `get a`, `wait 60`, `get a`, `get a`, `wait 20`, `get a`, `get b` →

```
get a: miss a#1 fetches=1
get a: hit a#1 fetches=1
  (waited 60 ms)
get a: stale a#1 fetches=2
get a: stale a#1 fetches=2
  (waited 20 ms)
get a: hit a#2 fetches=2
get b: miss b#3 fetches=3
```

The stale reads answered instantly with the old value while one refresh ran; the next read after it landed was fresh.

INPUT 1

```
40
get a
get a
wait 60
get a
get a
wait 20
get a
get b
```

OUTPUT 1

```
get a: miss a#1 fetches=1
get a: hit a#1 fetches=1
  (waited 60 ms)
get a: stale a#1 fetches=2
get a: stale a#1 fetches=2
  (waited 20 ms)
get a: hit a#2 fetches=2
get b: miss b#3 fetches=3
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =
const sleep = (ms, v) => new Promise((r) => setTimeout(r, ms, v));
let fetches = 0;
async function fetchValue(key) { fetches++; await sleep(10); return `${key}#${fetches}`;
class SwrCache {
  #entries = new Map();   // key -> { value, at, revalidating }
  constructor(ttlMs) { this.ttl = ttlMs; }
  async get(key) {
    const entry = this.#entries.get(key);
    if (!entry) {
      const value = await fetchValue(key);
      this.#entries.set(key, { value, at: Date.now(), revalidating: null });
      return { status: "miss", value };
    }
    if (Date.now() - entry.at < this.ttl) return { status: "hit", value: entry.value };
    if (!entry.revalidating) {                                    // stale: serve it
      entry.revalidating = fetchValue(key).then((value) => { Object.assign(entry, { value
    }
    return { status: "stale", value: entry.value };
  }
}
async function main() {
  const cache = new SwrCache(Number(lines[0]));
  for (const line of lines.slice(1)) {
    const [cmd, arg] = line.trim().split(/\s+/);
    if (cmd === "wait") { await sleep(Number(arg)); console.log(`  (waited ${arg} ms)`);
    else if (cmd === "get") { const r = await cache.get(arg); console.log(`get ${arg}: ${
  }
}
main();
```

CHECKPOINT QUESTIONS

1. `querySelectorAll` returns…

A. A live collection

B. A static `NodeList` snapshot — iterable, but not an array (spread it for `map`)

C. An array

D. The first match

2. Building a list of 1,000 items with one `appendChild` per item in a loop…

A. Is optimal

B. Can trigger repeated layout work — build in a `DocumentFragment` (or a template) and insert once

C. Is impossible

D. Reflows zero times

3. Which events do **not** bubble?

A. `click`, `input`

B. `focus` and `blur` (use `focusin` / `focusout`), `mouseenter` / `mouseleave`, element `scroll`

C. `submit`

D. `keydown`

4. `stopPropagation` in an inner handler to keep an outer one from running is…

A. The standard pattern

B. Usually a smell — it silently disables outer handlers (analytics, click-outside, delegation); prefer checking `e.target` in the outer handler

C. Required for forms

D. Faster

5. A `CustomEvent` carries its payload in…

A. `e.data`

B. `e.detail` — set via `new CustomEvent(type, { detail, bubbles })`

C. `e.payload`

D. `e.value`

6. A cross-origin `fetch` with `Content-Type: application/json` …

A. Is a simple request

B. Triggers a preflight `OPTIONS` request; the server must answer with the allowed methods and headers

C. Is blocked always

D. Needs no CORS

7. Storing a bearer token in `localStorage` …

A. Is the secure choice

B. Exposes it to any XSS on the origin; an `HttpOnly` cookie or in-memory storage is safer

C. Encrypts it

D. Sends it automatically

8. `SameSite=Lax` sends a cookie on a cross-site request only when…

A. Never

B. It is a top-level GET navigation — not on cross-site POSTs, iframes or fetches

C. Always

D. The request is HTTPS

9. A single-page app's server must…

A. Return 404 for unknown paths

B. Serve the app shell for every client route so deep links and refreshes work after `pushState`

C. Redirect to `/`

D. Disable caching

10. Layout thrashing is…

A. Too many DOM nodes

B. Interleaving DOM writes with layout reads so the browser reflows on every read — batch reads, then writes

C. Slow CSS

D. Large images

11. The most robust XSS defence for rich user HTML is…

A. A regex that strips `<script>`

B. An allow-list sanitiser (DOMPurify) plus a CSP and HttpOnly cookies — layers, never a deny-list regex

C. Lower-casing the input

D. `encodeURIComponent`

12. Subresource Integrity (`integrity="sha384-…"`) on a `<script src>` …

A. Speeds loading

B. Makes the browser refuse the script if its bytes do not match the hash — protection against a compromised CDN

C. Enables CORS

D. Minifies it

ANSWER KEY · CHECKPOINT — THE DOM AND THE BROWSER

1. B `getElementsByClassName` is the live one.

2. B `replaceChildren(fragment)` is one mutation.

3. B Delegating focus needs the bubbling variants.

4. B Legitimate uses exist but are rare.

5. B `dispatchEvent` returns `false` if a listener called `preventDefault` on a cancelable event.

6. B Form-encoded POSTs without custom headers are the "simple" case.

7. B Weigh convenience against the threat model.

8. B `Strict` never sends cross-site; `None` always (and requires `Secure`).

9. B Otherwise `/users/42` works only when reached by clicking.

10. B `IntersectionObserver` / `ResizeObserver` avoid measuring in handlers.

11. B Render plain text with `textContent` whenever HTML is not required.

12. B Third-party scripts run with full page privileges; pin, self-host, restrict with CSP.
