HTML SERIES · COURSE 3

HTML Advanced

Web Components, Canvas, SVG, drag-and-drop, the File API, offline storage, PWAs, deep accessibility, performance, and a final capstone. 12 chapters.

12 Chapters

osztromok.com

CHAPTER 1: WEB COMPONENTS IN DEPTH

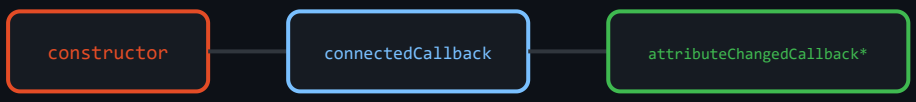
COURSE 3 · CH 1

Web Components in Depth

The full custom element lifecycle, templates feeding the shadow root, and slots for user-supplied content

Intermediate Course Chapter 11 introduced the core concept — a custom element with a shadow root. This chapter completes the picture: the full lifecycle a custom element actually goes through, building it properly with `<template>` instead of a string literal, and `<slot>` for letting users place their own content inside your component.

The Full Custom Element Lifecycle



*fires repeatedly, any time a watched attribute changes — not just once like the others

CALLBACK	FIRES WHEN
`constructor()`	The element instance is created — too early to safely read attributes or children yet
`connectedCallback()`	The element is actually inserted into the document (Intermediate Ch11) — the safe place to read attributes and build initial content
`disconnectedCallback()`	The element is removed from the document — clean up event listeners, timers, subscriptions here
`attributeChangedCallback(name, old, new)`	A watched attribute's value changes — only fires for attributes explicitly listed in observedAttributes

```
class RatingStars extends HTMLElement {
  static get observedAttributes() {
    return ['value']; // only "value" triggers attributeChangedCallback
  }

  attributeChangedCallback(name, oldValue, newValue) {
```

```
    if (name === 'value') {
      this.render(newValue);
    }
  }

  disconnectedCallback() {
    // clean up anything set up in connectedCallback, if applicable
  }
}
```

ATTRIBUTECHANGEDCALLBACK ONLY FIRES FOR ATTRIBUTES YOU EXPLICITLY LIST

Forgetting to add an attribute name to the static `observedAttributes` array means changes to it are silently ignored — the callback never fires for anything not in that list. A genuinely common first mistake when building a component that's supposed to react to attribute changes.

Using template with Shadow DOM — The Cleaner Pattern

Intermediate Chapter 11's example built shadow DOM content via a string template literal. A real component typically uses an actual `<template>` element (Intermediate Chapter 8) instead — genuine, validatable HTML rather than a string.

```
<template id="ratingStarsTemplate">
  <style>
    .star { color: gold; }
  </style>
  <span class="star">★★★★★</span>
</template>

<script>
class RatingStars extends HTMLElement {
  connectedCallback() {
    const template = document.getElementById('ratingStarsTemplate');
    const shadow = this.attachShadow({ mode: 'open' });
    shadow.appendChild(template.content.cloneNode(true));
  }
}
customElements.define('rating-stars', RatingStars);
</script>
```

slot — Letting Users Pass Their Own Content In

Without slots, a custom element's content is entirely fixed by its own internal template — there's no way for someone using the component to insert their own content into it. `<slot>` creates a placeholder the user's own light-DOM content gets projected into.

```
// Inside the component's shadow root template:
<div class="card">
  <h3>Card Title</h3>
  <slot></slot> // whatever the user puts inside <my-card> appears here
</div>

// Usage — this content gets projected into the slot above:
<my-card>
  <p>This is the user's own content, slotted in.</p>
</my-card>
```

Named Slots — Multiple Insertion Points

```
// Component template, with two distinct named slots:
<div class="card">
  <header><slot name="title"></slot></header>
  <slot></slot> // the unnamed default slot, for everything else
</div>

// Usage — slot="title" routes that specific element to the named slot above:
<my-card>
  <h3 slot="title">Card Title</h3>
  <p>Goes into the default slot instead.</p>
</my-card>
```

SLOTTED CONTENT KEEPS ITS ORIGINAL STYLING CONTEXT, UNLIKE THE REST OF THE SHADOW ROOT

Content placed into a slot is still technically "light DOM" — it's styled by the main page's CSS, not the component's internal shadow styles, even though it visually appears inside the component. This is a deliberate, useful exception to the otherwise strict shadow DOM encapsulation boundary from Intermediate Chapter 11.

CHAPTER 1 QUICK REFERENCE

- **Full lifecycle:** constructor → connectedCallback → attributeChangedCallback (repeatable) → disconnectedCallback

- **observedAttributes** must explicitly list every attribute that should trigger attributeChangedCallback
- **Build shadow content from a real <template>**, cloned via cloneNode(true), rather than a string literal
- **<slot>** — a placeholder for user-supplied light-DOM content to be projected into the shadow root
- **Named slots (slot="name")** — multiple distinct insertion points within one component
- **Slotted content keeps light-DOM styling** — a deliberate exception to shadow DOM's otherwise strict encapsulation
- **Next chapter:** Canvas API basics — drawing, animation fundamentals

CHAPTER 2: CANVAS API BASICS

COURSE 3 · CH 2

Canvas API Basics

Drawing pixel-based graphics directly with JavaScript, and the animation loop that makes them move

`<canvas>` is a blank rectangular surface that JavaScript draws onto pixel by pixel — shapes, images, text, entire game frames. Unlike every element covered so far, canvas content isn't part of the DOM at all once drawn; it's genuinely just pixels, with no individual shapes to inspect or style with CSS afterward.

Setting Up a Canvas

```
<canvas id="myCanvas" width="300" height="200"></canvas>

<script>
const canvas = document.getElementById('myCanvas');
const ctx = canvas.getContext('2d'); // the actual drawing API lives on this object
</script>
```

SET WIDTH/HEIGHT AS HTML ATTRIBUTES, NOT CSS, FOR THE ACTUAL DRAWING SURFACE SIZE

CSS width/height resize the canvas's display size, but the drawing surface's internal pixel resolution stays whatever the HTML attributes specified — mismatching the two stretches or blurs everything drawn. Set the real resolution via the HTML attributes; use CSS only if deliberately scaling the display size separately from that resolution.

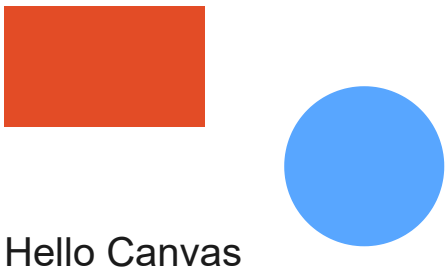
Basic Drawing

```
// A filled rectangle
ctx.fillStyle = '#e34c26';
ctx.fillRect(20, 20, 100, 60); // x, y, width, height

// A circle (canvas has no built-in circle — use an arc spanning the full 360°)
ctx.beginPath();
ctx.arc(200, 100, 40, 0, Math.PI * 2); // x, y, radius, startAngle, endAngle
ctx.fillStyle = '#58a6ff';
```

```
ctx.fill();

// Text
ctx.font = '20px sans-serif';
ctx.fillStyle = '#1a1a1a';
ctx.fillText('Hello Canvas', 20, 150);
```



The (0,0) Origin and Coordinate System

Canvas coordinates start at **(0,0) in the top-left corner**, with X increasing rightward and Y increasing *downward* — the opposite of typical mathematical graphing convention, but consistent with how most computer graphics systems work, including CSS positioning.

The Animation Loop — requestAnimationFrame

Animating on canvas means redrawing the entire frame repeatedly, slightly differently each time — `requestAnimationFrame` is the browser-native way to schedule that redraw at the right rate, synced to the display's actual refresh rate.

```
let x = 0;

function draw() {
  ctx.clearRect(0, 0, canvas.width, canvas.height); // erase the previous frame entirely

  ctx.fillStyle = '#3fb950';
  ctx.fillRect(x, 80, 40, 40);

  x += 2; // move slightly each frame
  if (x > canvas.width) x = 0; // wrap around

  requestAnimationFrame(draw); // schedule the next frame
```

```
}

requestAnimationFrame(draw); // kick off the loop
```

FORGETTING CLEARRECT LEAVES A SMEARED TRAIL OF EVERY PREVIOUS FRAME

Each call to `draw()` paints on top of whatever's already there — without clearing first, the moving square from the example above would leave a solid streak across the canvas rather than appearing to move cleanly. `clearRect` (or repainting the entire background) at the start of every frame is essentially mandatory for any genuine animation.

Canvas vs SVG — Choosing the Right Tool

🖼️ Canvas

Pixel-based — once drawn, individual shapes can't be selected, styled with CSS, or queried in the DOM. Better for many objects, frequent redraws, genuinely pixel-level work (image filters, particle effects, games).

📐 SVG (Chapter 3)

Vector/DOM-based — every shape is a real, individually addressable element, stylable with CSS, scales perfectly at any zoom level. Better for icons, diagrams, charts, and anything needing CSS styling or DOM-level interactivity per shape.

A ROUGH RULE OF THUMB: HUNDREDS OF SIMPLE SHAPES FAVOURS CANVAS, A FEW DOZEN COMPLEX ONES FAVOURS SVG

Canvas's pixel-based, no-DOM-overhead approach scales better to large numbers of objects redrawn every frame (particle systems, simple games). SVG's per-element DOM approach scales better when individual elements need their own styling, event handlers, or accessibility attributes — exactly the trade-off explored in full in Chapter 3.

CHAPTER 2 QUICK REFERENCE

- **canvas.getContext('2d')** — the object the actual drawing API lives on
- **Set width/height as HTML attributes**, not CSS, for the real drawing resolution
- **fillRect/arc/fillText** — basic shape and text drawing
- **Coordinate origin (0,0)** is top-left; Y increases downward
- **requestAnimationFrame** — schedules the next frame, synced to display refresh rate
- **clearRect at the start of every frame** — essential, or previous frames smear together

- **Canvas (pixels, many objects) vs SVG (DOM, stylable elements)** — pick based on object count and styling needs
- **Next chapter:** SVG in HTML — inline SVG, manipulating with CSS/JS

CHAPTER 3: SVG IN HTML

COURSE 3 · CH 3

SVG in HTML

Inline vector graphics that are genuinely part of the DOM — stylable with CSS, scriptable with JavaScript, perfectly scalable

Chapter 2 ended by contrasting canvas with SVG. This chapter delivers on that promise — SVG embedded directly in HTML, where every shape is a real DOM element you can select, style, and animate exactly like any other HTML element covered throughout this entire series.

Inline SVG — Genuinely Part of the Document

```
<svg viewBox="0 0 200 100" width="200" height="100">
  <rect x="10" y="10" width="80" height="50" fill="#e34c26"></rect>
  <circle cx="150" cy="35" r="25" fill="#58a6ff"></circle>
  <text x="10" y="85" font-size="14">Inline SVG</text>
</svg>
```

Inline SVG

Unlike canvas's pixel buffer, every `<rect>`, `<circle>`, and `<text>` here is a genuine, individually inspectable element in the DOM — visible and selectable in DevTools exactly like an ordinary `<div>` or `<p>`.

viewBox — The Key to Genuine Scalability

`viewBox="0 0 200 100"` defines the internal coordinate system (here, 200×100 units) independent of the actual rendered `width` / `height` — this is what lets an SVG scale to any size with zero blurring or pixelation, unlike a raster image or canvas content.

OMITTING VIEWBOX IS THE MOST COMMON REASON AN SVG "DOESN'T SCALE PROPERLY" WHEN RESIZED WITH CSS

> Without a viewBox, an SVG's internal coordinates are tied directly to its width/height attributes — resizing it via CSS afterward can crop or distort the content rather than scaling it cleanly. Setting a viewBox once, matching the original drawn dimensions, fixes this category of problem entirely.

Styling SVG with CSS — Exactly Like Any Other Element

```
<style>
  .my-icon { fill: #3fb950; transition: fill 0.2s; }
  .my-icon:hover { fill: #e34c26; }
</style>

<svg viewBox="0 0 24 24" width="24" height="24" class="my-icon">
  <circle cx="12" cy="12" r="10"></circle>
</svg>
```

Note `fill` as the CSS property controlling colour, not `color` or `background` — SVG has its own specific set of presentation properties (`fill`, `stroke`, `stroke-width`) that CSS can target directly, alongside the standard properties already familiar from the CSS courses.

Manipulating SVG with JavaScript

```
const circle = document.querySelector('circle');
circle.setAttribute('r', '40'); // grow the circle
circle.style.fill = '#a371f7'; // or set styles directly, exactly like any HTML element

circle.addEventListener('click', () => {
  console.log('Circle clicked!'); // real DOM events, per-shape
});
```

This per-element interactivity — a click handler on one specific shape, independent of every other shape on the page — is exactly what canvas content cannot offer at all, since canvas pixels have no individual identity once drawn.

SVG <img> vs Inline SVG

<img src="icon.svg">	**Inline <svg>…</svg>**

Simple, cacheable as a separate file, but the SVG's internals are completely opaque — no CSS styling of individual shapes, no JavaScript access from the main page at all.	Full CSS/JS access to every individual shape, but adds to the HTML file's size directly and isn't separately cacheable the way an external file is.

USE INLINE SVG SPECIFICALLY WHEN YOU NEED TO STYLE OR SCRIPT INDIVIDUAL PARTS

For a purely decorative icon that never changes, `<img src="icon.svg">` is simpler and perfectly adequate. For an icon that needs a hover colour change, an animated logo, or any shape-level interactivity, inline SVG is the only option that actually allows it.

Common SVG Elements Beyond the Basics

- **<path>** — the most powerful and common SVG element, drawing arbitrary curves and lines via a compact "d" attribute path-data string (the same underlying mechanism used by icon libraries and the kanji stroke animations elsewhere on osztromok.com).
- **<g>** — groups multiple shapes together, letting transforms and styles apply to the whole group at once.
- **<use>** — references and reuses a shape defined elsewhere, avoiding duplicating the same path data repeatedly.

CHAPTER 3 QUICK REFERENCE

- **Inline <svg>** — every shape is a real, inspectable DOM element, unlike canvas's opaque pixels
- **viewBox** — defines internal coordinates independent of rendered size; the key to genuine scaling without distortion
- **SVG presentation properties** (fill, stroke, stroke-width) — styled via CSS exactly like standard properties
- **setAttribute/style/addEventListener** — full per-shape JavaScript manipulation, impossible with canvas
- **<img src="x.svg">** for simple decorative icons; **inline SVG** when individual shapes need styling/scripting
- **<path>/<g>/<use>** — the building blocks behind most real-world icon sets and complex SVG graphics

- **Next chapter:** Drag and drop API

CHAPTER 4: DRAG AND DROP API

COURSE 3 · CH 4

Drag and Drop API

Native browser drag-and-drop — the event sequence, what dataTransfer actually carries, and where this API genuinely struggles

Dragging a file from the desktop into a browser, or reordering items in a list by dragging them — both rely on the same native HTML Drag and Drop API. It's older and has more rough edges than most APIs covered in this course, but understanding the actual event sequence makes it far less mysterious than it first appears.

Making an Element Draggable

```
<div draggable="true" id="item1">Drag me</div>

<div id="dropzone">Drop here</div>
```

`draggable="true"` is required on the source element — most elements default to non-draggable, with images and links being the only common exceptions that are draggable automatically.

The Full Event Sequence

*dragover fires repeatedly, continuously, while hovering over a valid drop target — not just once

EVENT	FIRES ON	PURPOSE
dragstart	The draggable source element	Set up dataTransfer with what's being dragged
dragover	Every potential drop target, repeatedly	Must call preventDefault() or the drop is rejected entirely

EVENT	FIRES ON	PURPOSE
drop	The actual drop target	Read dataTransfer, do the actual work
dragend	The original source element	Clean up — runs whether the drop succeeded or was cancelled

FORGETTING PREVENTDEFAULT() IN DRAGOVER IS THE SINGLE MOST COMMON REASON "NOTHING HAPPENS" ON DROP

By default, the browser rejects drops on almost everything — `dragover`'s handler must explicitly call `event.preventDefault()` to tell the browser "yes, this is a valid place to drop." Skipping this one line is responsible for the vast majority of "my drag and drop isn't working at all" debugging sessions.

dataTransfer — Carrying Information Between Source and Target

```
const source = document.getElementById('item1');
const dropzone = document.getElementById('dropzone');

source.addEventListener('dragstart', (event) => {
  event.dataTransfer.setData('text/plain', event.target.id);
});

dropzone.addEventListener('dragover', (event) => {
  event.preventDefault(); // required — see warning above
});

dropzone.addEventListener('drop', (event) => {
  event.preventDefault();
  const draggedId = event.dataTransfer.getData('text/plain');
  const draggedElement = document.getElementById(draggedId);
  event.target.appendChild(draggedElement); // actually move it in the DOM
});
```

`setData` / `getData` work with a MIME-type-like key, allowing several formats of the same data to be set simultaneously — useful when dragging between different applications (a browser tab and a desktop file manager) that each expect a different data format.

Dragging Files From the Desktop

```
dropzone.addEventListener('drop', (event) => {
  event.preventDefault();
```

```
  const files = event.dataTransfer.files; // a genuine FileList, same type as a file input
  for (const file of files) {
    console.log(file.name, file.size);
  }
});
```

Dragging files from the operating system into the browser populates `event.dataTransfer.files` with the same `FileList` type the File API (next chapter) works with directly — drag-and-drop file upload and a traditional `<input type="file">` ultimately produce the same kind of object to work with.

Where Native Drag and Drop Genuinely Struggles

- **Touch devices.** The native API is mouse-event-based at its core and doesn't translate cleanly to touch — mobile drag interactions typically need a separate, custom-built touch-event implementation.
- **Visual feedback during the drag.** Customising the default drag "ghost image," giving real-time visual feedback about valid/invalid drop zones, and smooth reordering animations all require considerable additional work beyond the bare API.
- **Accessibility.** Drag-and-drop interactions are inherently difficult for keyboard-only and screen reader users — a genuinely accessible interface needs an equivalent non-drag way to accomplish the same action (e.g. "Move up"/"Move down" buttons alongside drag handles).

FOR ANYTHING BEYOND A SIMPLE FILE-DROP ZONE, MANY REAL PROJECTS REACH FOR A LIBRARY

Reordering lists, Kanban boards, and similar complex drag interactions are commonly built with a dedicated library (SortableJS and similar) rather than the raw native API directly — they handle touch support, accessibility fallbacks, and animation smoothing that the bare API leaves entirely up to you. Worth knowing the native API exists and how it fundamentally works, even when a library ends up handling the actual implementation.

CHAPTER 4 QUICK REFERENCE

- **draggable="true"** — required on the source element; images/links are draggable by default
- **Event sequence:** dragstart → dragover (repeatedly) → drop → dragend

- **dragover MUST call preventDefault()** — otherwise the drop is rejected by default; the most common bug
- **dataTransfer.setData/getData** — carries data between source and target, keyed by a MIME-type-like string
- **event.dataTransfer.files** — a real FileList when files are dragged from the desktop, same type as a file input
- **Native API struggles with:** touch devices, custom drag visuals, accessibility — often handled via a dedicated library instead
- **Next chapter:** File API — file inputs, reading files client-side

CHAPTER 5: FILE API

COURSE 3 · CH 5

File API

Reading a file's contents entirely client-side — image previews, text parsing, and validation, all before anything uploads

Chapter 4 ended with dragged files landing in `dataTransfer.files` as a `FileList` — the exact same type a regular `<input type="file">` (Intermediate Chapter 1) produces. This chapter covers what you can actually do with that `File` object once you have it, entirely in the browser, with no server round-trip required at all.

Getting Files from an Input

```
<input type="file" id="fileInput" accept="image/*">

<script>
document.getElementById('fileInput').addEventListener('change', (event) => {
  const file = event.target.files[0]; // files is a FileList, even for a single file
  console.log(file.name, file.size, file.type);
});
</script>
```

file.name
The original filename

file.size
Size in bytes

file.type
MIME type, e.g. "image/png"

file.lastModified
Timestamp of last modification

ACCEPT DOESN'T ACTUALLY RESTRICT WHAT A USER CAN SELECT

The `accept` attribute only filters the OS file picker dialog's default view — most file pickers still let a user manually choose "All Files" and pick anything regardless. Always re-check `file.type` in JavaScript (and validate the actual file content server-side, never trust the client-reported type alone) rather than relying on `accept` as genuine validation.

FileReader — Reading the Actual Content

A `File` object alone only gives metadata — `FileReader` actually reads the bytes, asynchronously, in one of several formats depending on what's needed.

METHOD	READS THE FILE AS
`readAsDataURL()`	A base64 data URL — directly usable as an <img> src, for instant previews
`readAsText()`	Plain text — for .txt, .csv, .json files
`readAsArrayBuffer()`	Raw binary data — for processing binary formats directly

A Real Example — Instant Image Preview Before Upload

```
document.getElementById('fileInput').addEventListener('change', (event) => {
  const file = event.target.files[0];
  if (!file) return;

  const reader = new FileReader();

  reader.onload = () => {
    const img = document.getElementById('preview');
    img.src = reader.result; // the data URL, ready to display directly
  };

  reader.readAsDataURL(file);
});
```

This is exactly the pattern behind every "see a preview before you upload" experience — the user picks a file, sees it rendered instantly, and nothing has been sent to a server at all yet. The actual upload (if any) happens separately, typically via the `FormData` covered in Intermediate Chapter 10.

Reading Text Files — Client-Side CSV/JSON Preview

```
reader.onload = () => {
  const text = reader.result;
  try {
    const data = JSON.parse(text);
    console.log('Parsed JSON:', data);
```

```
    } catch (err) {
      console.error('Not valid JSON');
    }
  };
  reader.readAsText(file);
```

Drag-and-Drop + File API — Tying Chapters 4 and 5 Together

```
dropzone.addEventListener('drop', (event) => {
  event.preventDefault();
  const file = event.dataTransfer.files[0]; // Chapter 4's dataTransfer.files
  const reader = new FileReader(); // Chapter 5's FileReader, same as a normal input
  reader.onload = () => { /* preview, parse, etc. — identical to above */ };
  reader.readAsDataURL(file);
});
```

Because a dragged file and a file selected via `<input type="file">` both produce identical `File` objects, the exact same `FileReader` code handles both without any special-casing — a genuinely clean piece of API design that pays off whenever both interaction methods are offered together.

READING A FILE CLIENT-SIDE IS NEVER A SUBSTITUTE FOR SERVER-SIDE VALIDATION ON UPLOAD

Everything in this chapter happens entirely in the user's own browser and tells you nothing trustworthy about what actually arrives at your server — a malicious user can trivially fabricate a request with completely different file content than whatever the client-side code displayed or validated. Treat client-side file reading purely as a UX convenience (instant previews, early format checks); always re-validate file type, size, and content again on the server before trusting or storing anything.

CHAPTER 5 QUICK REFERENCE

- **input.files** — a FileList, even for a single file; access individual files by index
- **accept attribute doesn't enforce anything** — just filters the picker's default view; always re-check file.type in JS
- **FileReader.readAsDataURL()** — base64 data URL, directly usable as an img src for instant previews
- **readAsText() / readAsArrayBuffer()** — for text content and raw binary data respectively
- **reader.onload** fires asynchronously once reading completes; result available as reader.result
- **Drag-and-drop files and input-selected files produce identical File objects** — the same FileReader code handles both

- **Client-side reading is UX only** — server-side validation of every uploaded file remains essential regardless
- **Next chapter:** Web Storage & offline — localStorage/IndexedDB, service workers intro

CHAPTER 6: WEB STORAGE & OFFLINE

COURSE 3 · CH 6

Web Storage & Offline

A quick storage recap, IndexedDB in actual use, and a first look at service workers — the foundation of offline-capable pages

CROSS-REFERENCE, NOT A REPEAT

Cookies, localStorage, sessionStorage, and the conceptual comparison with IndexedDB were already covered thoroughly in the separate Browser Storage, Cookies & Security course's first chapter. This chapter assumes that comparison and goes straight to using IndexedDB practically, plus service workers — the piece that course didn't cover, since it's specifically an HTML/web-app capability rather than a browser-security topic.

IndexedDB in Practice

IndexedDB's API is genuinely more verbose than localStorage's simple get/set — it's asynchronous and event-based, reflecting that it's a real database, not a quick key-value store.

```
const request = indexedDB.open('NotesApp', 1);

request.onupgradeneeded = (event) => {
  const db = event.target.result;
  db.createObjectStore('notes', { keyPath: 'id' }); // runs only on first creation or version b
};

request.onsuccess = (event) => {
  const db = event.target.result;

  // Writing a record
  const tx = db.transaction('notes', 'readwrite');
  tx.objectStore('notes').add({ id: 1, text: 'Buy milk' });

  // Reading it back
  const getRequest = db.transaction('notes').objectStore('notes').get(1);
  getRequest.onsuccess = () => console.log(getRequest.result);
};
```

- **onupgradeneeded** fires only when the database is first created, or when the version number is bumped — exactly where schema changes (adding object stores, indexes) belong.
- **Object stores** are roughly analogous to tables — each with a `keyPath` defining what makes a record unique.
- **Transactions** wrap every read/write — IndexedDB never lets you read or write outside one, ensuring consistency even with concurrent access.

MOST REAL PROJECTS USE A WRAPPER LIBRARY RATHER THAN THE RAW INDEXEDDB API DIRECTLY

The verbosity and callback-heavy style above is exactly why libraries like idb (a thin Promise-based wrapper) are extremely common in practice — they provide the same underlying storage with a far more ergonomic, modern async/await-friendly interface. Worth understanding the raw API conceptually, as shown here, even if a wrapper ends up handling the actual implementation in a real project.

Service Workers — A Programmable Network Proxy

A service worker is a script that runs separately from any page, sitting between your web app and the network — it can intercept every network request a page makes and decide how to respond, including serving a cached response when genuinely offline.

The service worker decides, per request, whether to answer from cache or actually go to the network — even fully offline, with no network at all

A Minimal Service Worker — Registration and Caching

```
// In your main page's JavaScript:
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js');
}

// sw.js — the service worker script itself, runs separately
const CACHE_NAME = 'my-app-v1';

self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME).then((cache) =>
```

```
      cache.addAll(['/', '/styles.css', '/app.js'])
    )
  );
});

self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request).then((cached) => cached || fetch(event.request))
  );
});
```

The `fetch` handler here implements a simple "cache first" strategy: check the cache, serve it if present, fall back to the real network only if not. With this in place, the listed files remain available even with no network connection at all.

SERVICE WORKERS REQUIRE HTTPS (WITH ONE EXCEPTION)

Browsers refuse to register a service worker over plain HTTP, since it has powerful, security-sensitive capabilities (intercepting all network requests) — the one exception is `localhost`, which is treated as secure for development purposes. This connects directly back to the security course's HTTPS coverage — yet another reason HTTPS isn't optional for a modern site.

THIS IS GENUINELY JUST THE FOUNDATION — CHAPTER 7 BUILDS THE FULL PWA PICTURE

A service worker plus a web app manifest (covered next chapter) together make a page genuinely installable, with an icon, offline support, and an app-like presentation — this chapter's caching example is the simplest possible starting point that real Progressive Web Apps build much further on top of.

CHAPTER 6 QUICK REFERENCE

- **IndexedDB** — asynchronous, transaction-based; object stores ≈ tables, keyPath defines uniqueness
- **onupgradeneeded** — fires only on first creation or version bump; the place for schema changes
- **Most real projects use a wrapper library** (e.g. idb) over the raw, verbose IndexedDB API directly
- **Service worker** — a separate script intercepting network requests, enabling offline behaviour
- **install event + caches.open/addAll** — pre-caching essential files
- **fetch event + caches.match** — deciding cache-vs-network per request, the basis of offline support
- **Requires HTTPS** (localhost excepted) — same security principle as the cookies/security course

- **Next chapter:** Progressive Web Apps — manifest.json, installability basics

CHAPTER 7: PROGRESSIVE WEB APPS

COURSE 3 · CH 7

Progressive Web Apps

manifest.json and the specific requirements that make a web page genuinely installable, with an icon and an app-like feel

Chapter 6's service worker is one half of what makes a Progressive Web App (PWA) — the other half is the **web app manifest**, a JSON file describing how the app should behave and present itself once installed. Together, they're what turns "just a website" into something a user can add to their home screen and open like a native app.

The Web App Manifest

```
// manifest.json
{
  "name": "My Learning Site",
  "short_name": "Learning",
  "start_url": "/",
  "display": "standalone",
  "background_color": "#0d1117",
  "theme_color": "#e34c26",
  "icons": [
    { "src": "/icons/icon-192.png", "sizes": "192x192", "type": "image/png" },
    { "src": "/icons/icon-512.png", "sizes": "512x512", "type": "image/png" }
  ]
}
```

```
// Linked from the page's <head>, like a stylesheet
<link rel="manifest" href="/manifest.json">
```

PROPERTY	EFFECT
`name` / `short_name`	Full app name vs the shorter version shown under the home screen icon
`start_url`	The URL opened when launched from the installed icon

PROPERTY	EFFECT
`display`	standalone hides browser UI (address bar, tabs) — the app-like presentation that's the whole point
`theme_color`	Colours the OS status bar/title bar to match your brand on supporting platforms
`icons`	Multiple sizes — different OS/launcher contexts request different resolutions

The display Property — Choosing How "App-Like" It Feels

- **fullscreen** — hides absolutely everything, including the OS status bar; mainly suited to games.
- **standalone** — hides browser UI but keeps the OS status bar; the most common choice, looks genuinely like a native app.
- **minimal-ui** — keeps a minimal set of browser navigation controls visible.
- **browser** — opens as a completely normal browser tab, the default if no manifest exists at all.

What Actually Makes a Page Genuinely Installable

Browsers don't show the "Install" prompt for just any page with a manifest file — there's a specific, real checklist that must be satisfied first.

- ☑ **Served over HTTPS**
  Same requirement as service workers (Chapter 6) — no exceptions besides localhost during development.
- ☑ **A valid manifest.json, linked from the page**
  Including at minimum a name, icons of the required sizes, and a start_url.
- ☑ **A registered service worker with at least a fetch event handler**
  Chapter 6's minimal example already satisfies this — the browser checks that the page can genuinely respond to network requests, even if just passing them through.
- ☑ **Icons of the correct minimum sizes (typically 192×192 and 512×512)**
  Different platforms request different sizes — providing both common sizes covers the vast majority of cases.

MEETING THE CHECKLIST DOESN'T GUARANTEE AN AUTOMATIC INSTALL PROMPT APPEARS

Browsers use their own internal heuristics (engagement signals, how the site has been used) on top of the technical checklist before showing a spontaneous "Add to Home Screen" prompt — meeting every requirement is necessary but not always sufficient for the prompt to appear automatically. Most real sites also offer their own explicit "Install App" button, triggered via the `beforeinstallprompt` event, rather than relying purely on the browser's own judgement.

Triggering Your Own Install Prompt

```
let deferredPrompt;

window.addEventListener('beforeinstallprompt', (event) => {
  event.preventDefault(); // stop the browser's automatic prompt
  deferredPrompt = event; // save it to trigger manually later
  document.getElementById('installButton').hidden = false;
});

document.getElementById('installButton').addEventListener('click', () => {
  deferredPrompt.prompt(); // shows the real browser install UI, on your own button click
});
```

PWA SUPPORT AND BEHAVIOUR GENUINELY VARIES BY PLATFORM

iOS Safari historically supported PWAs with notably more limitations than Android Chrome (different install flow, fewer manifest properties honoured, no automatic install prompt at all on iOS) — testing on the actual target platforms matters far more for PWAs than for most other features covered in this course.

CHAPTER 7 QUICK REFERENCE

- **manifest.json** — name, icons, start_url, display mode; linked via <link rel="manifest">
- **display: standalone** — the most common choice; hides browser UI, keeps OS status bar
- **Installability checklist:** HTTPS, valid manifest, registered service worker with fetch handler, correctly sized icons
- **Meeting the checklist ≠ guaranteed automatic prompt** — browsers add their own engagement heuristics on top
- **beforeinstallprompt + prompt()** — trigger your own explicit "Install" button instead of relying on the browser alone

- **PWA support varies significantly by platform** — test on actual target devices, especially iOS vs Android
- **Next chapter:** accessibility deep dive — full ARIA patterns, keyboard navigation, screen reader testing

CHAPTER 8: ACCESSIBILITY DEEP DIVE

COURSE 3 · CH 8

Accessibility Deep Dive

Full ARIA widget patterns, keyboard navigation beyond Tab/Enter, and how to actually test with a real screen reader

Intermediate Chapter 6 covered ARIA's first rule and the most common attributes. This chapter goes into genuinely complex interactive widgets — tabs, accordions — where building correct keyboard and screen reader behaviour requires deliberate, specific patterns rather than relying on a native element doing the work automatically.

The ARIA Authoring Practices Guide — The Reference Worth Knowing About

The W3C maintains a living reference (the ARIA Authoring Practices Guide, "APG") with exact, tested patterns for every common complex widget — tabs, accordions, comboboxes, menus, dialogs. Rather than inventing keyboard behaviour from scratch, checking the APG's published pattern for a given widget type first avoids reinventing something that's already been carefully specified and tested.

The Tabs Pattern

```
<div role="tablist" aria-label="Settings sections">
  <button role="tab" aria-selected="true" aria-controls="panel-general" id="tab-general">Genera
  <button role="tab" aria-selected="false" aria-controls="panel-privacy" id="tab-privacy" tabin
</div>

<div role="tabpanel" id="panel-general" aria-labelledby="tab-general">...</div>
<div role="tabpanel" id="panel-privacy" aria-labelledby="tab-privacy" hidden>...</div>
```

Roving tabindex — The Key Technique Behind Tabs

Notice only the selected tab has `tabindex="0"` (the default for a button) while the unselected one has `tabindex="-1"` — this is the "roving tabindex" pattern: exactly one item in the group is reachable by Tab at any time, while Left/Right arrow keys (handled by your own JavaScript) move selection and update which tab has `tabindex="0"`.

WITHOUT ROVING TABINDEX, TAB WOULD STOP ON EVERY SINGLE TAB INDIVIDUALLY

Without this pattern, a keyboard user pressing Tab would land on each tab button one at a time before ever reaching the actual content — genuinely tedious for a row of even five or six tabs. Roving tabindex makes the whole tablist a single Tab stop, with arrow keys handling movement within it, matching how a native `<select>` already behaves.

The Accordion Pattern

```
<h3>
  <button aria-expanded="false" aria-controls="section1-content">Section 1</button>
</h3>
<div id="section1-content" hidden>...</div>
```

RECALL INTERMEDIATE COURSE 2'S DETAILS/SUMMARY — OFTEN THE SIMPLER CHOICE

A genuine accordion with this exact ARIA pattern is appropriate when custom styling/behaviour beyond what `<details>` / `<summary>` offers is genuinely required. For a straightforward expand/collapse FAQ, the native element from Intermediate Chapter 8 gets all of this correct automatically, with far less code — worth checking whether the simpler native option is sufficient before building a custom ARIA pattern.

Common ARIA Widget Patterns at a Glance

PATTERN	KEY ROLES/ATTRIBUTES	KEY KEYBOARD BEHAVIOUR
Tabs	tablist/tab/tabpanel, aria-selected	Arrow keys move between tabs, roving tabindex
Accordion	aria-expanded, aria-controls	Enter/Space toggles; often just native button behaviour
Menu	menu/menuitem, aria-expanded on the trigger	Arrow keys navigate items, Escape closes

PATTERN	KEY ROLES/ATTRIBUTES	KEY KEYBOARD BEHAVIOUR
`Combobox`	combobox/listbox/option, aria-expanded, aria-activedescendant	Arrow keys move through suggestions, Enter selects

Testing with a Real Screen Reader

NVDA (Windows, free)

The most widely used free option for testing on Windows — genuinely worth installing and learning the basics (NVDA key + arrows for browsing) rather than relying purely on DevTools' accessibility tree.

VoiceOver (macOS/iOS, built in)

Built into every Mac and iPhone — Cmd+F5 toggles it on macOS. No separate install needed, making it the lowest-friction way to do a first real screen reader test.

JAWS (Windows, commercial)

The most commonly used screen reader among professional/enterprise users specifically — paid, but worth knowing it exists and behaves slightly differently from NVDA in some edge cases.

A Minimal Testing Routine

- **Turn on a screen reader and close your eyes (or turn off the monitor).** Genuinely try to complete the page's main task using only what you hear.
- **Navigate by headings first** (most screen readers have a dedicated heading-jump shortcut) — confirms the heading structure from Fundamentals Chapter 2 actually makes sense out loud.
- **Tab through every interactive element** — confirm each one announces a clear name and role, and that focus order matches the visual reading order.
- **Try the actual complex widgets** (any tabs, accordions, custom dropdowns) specifically — this is where most real accessibility bugs concentrate.

A CUSTOM WIDGET THAT "LOOKS RIGHT" CAN STILL BE COMPLETELY SILENT OR CONFUSING TO A SCREEN READER

Visual correctness and accessible correctness are genuinely independent — a beautifully styled custom dropdown built entirely from divs with click handlers, with zero ARIA attributes, can be entirely invisible to a

screen reader user, announcing nothing useful at all despite working perfectly for a sighted mouse user. This is exactly why manual screen reader testing remains necessary even after automated tools (Intermediate Ch6) pass cleanly.

CHAPTER 8 QUICK REFERENCE

- **ARIA Authoring Practices Guide (APG)** — the W3C's tested reference patterns; check it before inventing custom widget behaviour
- **Roving tabindex** — exactly one item reachable by Tab at a time; arrow keys move within the group (tabs, menus)
- **Prefer details/summary over a custom accordion** when the native element's behaviour is genuinely sufficient
- **Common patterns:** tabs, accordion, menu, combobox — each with specific roles and keyboard expectations
- **Real screen readers:** NVDA (Windows, free), VoiceOver (Mac/iOS, built in), JAWS (Windows, commercial)
- **Minimal testing routine:** heading navigation, full Tab traversal, specifically exercise any custom widgets
- **Visual correctness ≠ accessible correctness** — manual screen reader testing catches what automated tools and a sighted check both miss
- **Next chapter:** performance — lazy loading, preload/prefetch, critical rendering path

CHAPTER 9: PERFORMANCE

COURSE 3 · CH 9

Performance

The critical rendering path, lazy loading, and resource hints — the HTML-level techniques that genuinely affect load speed

Most performance advice focuses on JavaScript and CSS optimisation — this chapter covers what HTML markup itself can do, often with a single attribute, to measurably improve how quickly a page becomes usable.

The Critical Rendering Path — What the Browser Actually Does First

A <script> tag (without async/defer) encountered mid-parse BLOCKS this entire sequence until it finishes downloading and executing

The browser parses HTML top to bottom, building the DOM as it goes — but a plain `<script>` tag stops everything until that script downloads and runs, which is exactly why blocking scripts in `<head>` were historically such a common performance problem.

```
// Blocks parsing until downloaded AND executed — avoid in <head> for non-critical scripts
<script src="analytics.js"></script>

// Downloads in parallel with parsing, executes in document order once ready
<script src="app.js" defer></script>

// Downloads in parallel, executes immediately whenever it finishes — order not guaranteed
<script src="widget.js" async></script>
```

DEFER VS ASYNC — THEY SOLVE GENUINELY DIFFERENT PROBLEMS

`defer` guarantees scripts run in document order, after parsing completes — the right default for most app code that depends on the DOM or on running in a specific sequence. `async` makes no ordering guarantee at all — appropriate for independent, self-contained scripts (a standalone analytics snippet) that don't depend on anything else or need to run at a specific point.

Lazy Loading — Deferring Off-Screen Content

```
<img src="photo.jpg" loading="lazy" alt="...">
<iframe src="..." loading="lazy" title="..."></iframe>
```

A single attribute defers loading until the element is about to scroll into view — already mentioned for iframes in Intermediate Chapter 4, and equally valuable on images, especially for a long page with many images further down that most visitors never reach.

NEVER LAZY-LOAD THE IMAGE VISIBLE IMMEDIATELY ON PAGE LOAD

Applying `loading="lazy"` to a hero image or anything visible without scrolling actively delays it unnecessarily — the browser still has to detect it's "in view" before starting the load, adding latency to exactly the content a visitor sees first. Reserve lazy loading specifically for genuinely below-the-fold content.

Resource Hints — preload, prefetch, preconnect

HINT	TELLS THE BROWSER TO	USE FOR
preload	Fetch this resource NOW, with high priority — it's needed very soon	A critical font or hero image the page needs immediately
prefetch	Fetch this resource at low priority, for likely future use	A resource needed on the NEXT page a user will probably visit
preconnect	Establish the connection (DNS, TCP, TLS) to a domain early, without fetching anything yet	A third-party domain (CDN, API) you know you'll request from soon

```
<link rel="preload" href="/fonts/main.woff2" as="font" type="font/woff2" crossorigin>
<link rel="preconnect" href="https://fonts.googleapis.com">
<link rel="prefetch" href="/next-page-assets.js">
```

PRELOAD IS SPECIFICALLY FOR THIS PAGE; PREFETCH IS A BET ON THE NEXT ONE

Using preload for something not genuinely needed immediately wastes bandwidth competing with resources that actually are urgent — the browser takes the priority hint seriously. Reserve preload for what this exact page needs right away; use prefetch when there's a real, well-justified expectation of where the user is headed next (the obvious "next chapter" link in a course, for instance).

Why HTML-Level Performance Still Matters Alongside JS/CSS Optimisation

- **These changes require no build tooling.** A single attribute or link tag, no bundler configuration, no framework-specific optimisation plugin.
- **They affect the very earliest part of the load,** before any JavaScript has even had a chance to run — exactly the window where a slow, render-blocking resource costs the most in perceived performance.
- **They're measurable directly** via the Network tab's waterfall view and Lighthouse's performance audit, both of which flag missing lazy-loading and render-blocking resources explicitly.

CHAPTER 9 QUICK REFERENCE

- **Critical rendering path:** parse HTML → build DOM → build CSSOM → render/paint; a blocking script stops this entire sequence
- **defer** — runs in document order after parsing; **async** — runs immediately on completion, no ordering guarantee
- **loading="lazy"** on images/iframes — defer off-screen content; never apply to anything visible without scrolling
- **preload** — high priority, needed now; **prefetch** — low priority, needed on a likely next page; **preconnect** — early connection setup only
- **These are HTML-level, zero-build-tooling wins** — measurable directly via Network waterfall and Lighthouse
- **Next chapter:** HTML5 APIs grab-bag — Geolocation, Notifications, Clipboard API

CHAPTER 10: HTML5 APIS GRAB-BAG

COURSE 3 · CH 10

HTML5 APIs Grab-Bag

Geolocation, Notifications, and the Clipboard API — three browser-native capabilities, all permission-gated, all genuinely useful in the right context

Several browser APIs don't fit neatly into any single theme covered so far, but share one important trait: each requires explicit user permission before doing anything, since each touches something genuinely sensitive (location, push interruptions, clipboard contents).

Geolocation API

```
navigator.geolocation.getCurrentPosition(
  (position) => {
    console.log(position.coords.latitude, position.coords.longitude);
  },
  (error) => {
    console.error('Location denied or unavailable:', error.message);
  }
);
```

📍 getCurrentPosition()

One-off location request — triggers the browser's permission prompt the first time, then returns coordinates via the success callback.

🔄 watchPosition()

Continuously tracks location as it changes, calling the success callback repeatedly — for genuine live-tracking use cases, not a one-time lookup.

GEOLOCATION REQUIRES HTTPS — YET ANOTHER ITEM ON THIS COURSE'S GROWING LIST

Same requirement as service workers and PWA installability (Chapters 6-7) — browsers refuse geolocation access on plain HTTP (localhost excepted), since location data is genuinely sensitive. By this point in the course, the pattern should be clear: meaningful, security-sensitive browser capabilities consistently require HTTPS.

Notifications API

```
// Must request permission explicitly — never assume it's already granted
Notification.requestPermission().then((permission) => {
  if (permission === 'granted') {
    new Notification('New message', {
      body: 'You have a new message from Philip',
      icon: '/icon.png'
    });
  }
});
```

Three possible permission states: `granted`, `denied`, and `default` (not yet asked) — once a user explicitly denies, most browsers don't re-prompt automatically; the user has to manually re-enable it via browser settings.

ASKING FOR NOTIFICATION PERMISSION IMMEDIATELY ON PAGE LOAD IS A NEAR-UNIVERSALLY DISLIKED PATTERN

A permission prompt appearing before a user has done anything, with no context for why notifications would be useful, is one of the most commonly cited "annoying website behaviour" complaints — and often results in an instant, permanent denial. Requesting permission in direct response to a relevant user action (clicking "Notify me when this is back in stock") converts far better and respects the user's context.

Clipboard API

```
// Writing to the clipboard — must be triggered by a genuine user action (a click), not run fre
document.getElementById('copyButton').addEventListener('click', () => {
  navigator.clipboard.writeText('https://osztromok.com/linux')
    .then(() => console.log('Copied!'));
});

// Reading from the clipboard — also gated, and often needs explicit permission
navigator.clipboard.readText().then((text) => {
  console.log('Clipboard contains:', text);
});
```

THE CLASSIC "COPY LINK" BUTTON PATTERN

A button that copies a URL or code snippet to the clipboard, with a brief "Copied!" confirmation message, is one of the most common and genuinely useful real-world applications of this API — and exactly the kind of feature a course-content site like osztromok.com could reasonably use for sharing a specific chapter's link.

CLIPBOARD WRITES MUST ORIGINATE FROM A GENUINE USER GESTURE

Browsers block clipboard-writing calls that aren't triggered directly by a user action like a click — a script that tries to silently write to the clipboard on page load, on a timer, or in response to a non-interactive event will simply fail. This is a deliberate security restriction, preventing pages from silently overwriting whatever a user has copied without their knowledge.

The Common Thread — Permissions and Graceful Fallbacks

Every API in this chapter can fail — denied permission, unsupported browser, an insecure context. Real code should always check for support and handle the failure path explicitly, never assume success.

```
if ('geolocation' in navigator) {
  // safe to use
} else {
  // offer a fallback — e.g. let the user type in their location manually
}
```

CHAPTER 10 QUICK REFERENCE

- **Geolocation:** getCurrentPosition() for a one-off lookup, watchPosition() for continuous tracking; requires HTTPS
- **Notifications:** requestPermission() returns granted/denied/default; ask in response to a relevant user action, never on page load
- **Clipboard:** writeText()/readText(); writes must originate from a genuine user gesture (a click), not run freely
- **Common pattern across all three:** explicit permission required, can fail, always handle the failure path gracefully
- **Check for API support** ('geolocation' in navigator, etc.) before relying on any of them
- **Next chapter:** Internationalization — lang attributes, dir, character encoding edge cases

CHAPTER 11: INTERNATIONALIZATION

COURSE 3 · CH 11

Internationalization

lang attributes done properly, right-to-left text direction, and the character encoding edge cases that actually bite

Fundamentals Chapter 1 introduced `lang="en"` on `<html>` as a quick rule. This chapter covers why it matters beyond a checkbox, how to mark mixed-language content correctly, and the encoding details that explain real, occasionally baffling text-rendering bugs.

lang — More Than One Attribute, More Than Once

```
<html lang="en"> // the document's overall, dominant language

// A specific phrase in a different language, within otherwise-English content
<p>The French phrase <span lang="fr">je ne sais quoi</span> has no direct English equivalent.</
```

- **Screen readers switch pronunciation** for the inner `lang`-tagged span, reading the French phrase with French phonetics rather than mispronouncing it as English text.
- **Browser spell-check** stops flagging the foreign phrase as a misspelling within an otherwise English document.
- **CSS can target language** via the `:lang()` pseudo-class, for language-specific typography rules (quotation mark styles genuinely differ between languages).

A MISSING OR WRONG LANG ATTRIBUTE HAS REAL, COMPOUNDING CONSEQUENCES

Beyond the immediate screen reader mispronunciation, search engines use the lang attribute to serve the right version of a page to the right regional audience, and translation tools use it to decide whether (and how) to offer a translation prompt at all. A small, easily-forgotten attribute with surprisingly wide-reaching downstream effects.

Hreflang — Multiple Language Versions of the Same Content

```
<link rel="alternate" hreflang="en" href="https://osztromok.com/en/linux">
<link rel="alternate" hreflang="hu" href="https://osztromok.com/hu/linux">
```

For a site genuinely offering the same content in multiple languages at different URLs, `hreflang` link tags (in `<head>`) tell search engines which version to show users searching in each specific language — directly relevant should osztromok.com ever offer translated versions of its Hungarian-language-learning content, for instance.

Right-to-Left Text — the dir Attribute

```
<p dir="rtl"lang="ar">مرحبا بالعالم</p>
```

Arabic text, rendered right-to-left, demonstrating how dir="rtl" reverses the natural reading and layout direction. — مرحبا بالعالم

Languages like Arabic and Hebrew read right-to-left — `dir="rtl"` doesn't just reverse the text itself (the browser already handles that based on the actual characters), it reverses the entire layout flow: where padding/margins effectively apply, which side elements align to, the direction lists indent.

> **DIR="AUTO" LETS THE BROWSER DETECT DIRECTION FROM THE ACTUAL CONTENT**
>
> For genuinely mixed or user-generated content where the language isn't known in advance (a comment box that might receive English or Arabic text), `dir="auto"` inspects the first strongly-directional character actually typed and sets direction automatically — useful specifically for that unpredictable-content case, though an explicit `dir` value is preferable whenever the language is actually known ahead of time.

Character Encoding — Where Real Bugs Actually Come From

Fundamentals Chapter 1 established `<meta charset="UTF-8">` as the modern standard. The edge cases worth understanding: what actually goes wrong, and why, when encoding is mismatched.

SYMPTOM	LIKELY CAUSE
`Garbled characters (mojibake), e.g. "â€™" instead of an apostrophe`	A file saved/read in a different encoding than declared — classic mismatch between actual bytes and the declared charset
`Question marks or boxes (□) replacing some characters`	The character genuinely doesn't exist in the encoding/font being used to render it
`Correct in the editor, garbled in the browser`	charset meta tag missing or declared after too much content, or the actual file saved in a different encoding than the meta tag claims

THE CHARSET META TAG MUST BE AMONG THE VERY FIRST BYTES OF THE DOCUMENT — A RULE WORTH RESTATING FROM FUNDAMENTALS CHAPTER 1

Browsers read a limited number of initial bytes looking for the encoding declaration before they start interpreting the rest of the content — if the charset declaration comes too late (after a large comment block, for instance), the browser may have already started parsing with a guessed, potentially wrong encoding.

THE KANJI COURSE'S "KANJI_父.HTML" STYLE FILENAMES ARE A REAL, PRACTICAL UTF-8 SUCCESS STORY

Using actual Unicode characters directly as filenames (rather than transliterated or romanised names) works correctly specifically because the whole stack — filesystem, server, browser — consistently treats everything as UTF-8 throughout. The same consistency that prevents the garbling symptoms above is exactly what makes a filename like that work reliably at all.

CHAPTER 11 QUICK REFERENCE

- **lang** — set on <html> for the document, and on individual elements for foreign-language phrases within otherwise different-language content
- **lang affects:** screen reader pronunciation, spell-check, CSS :lang() targeting, search engine regional serving
- **hreflang link tags** — declare alternate-language versions of the same content at different URLs
- **dir="rtl"** — reverses entire layout flow, not just text direction, for right-to-left languages
- **dir="auto"** — detects direction from actual content; useful for unpredictable user-generated text
- **Mojibake/garbled text** — almost always an encoding mismatch between actual bytes and declared/assumed charset

- **charset meta tag must be among the very first bytes** — browsers guess encoding from early content if it's declared too late
- **Next chapter (final, capstone):** building an accessible, performant, semantic multi-feature page

CHAPTER 12: CAPSTONE - MULTI-FEATURE PAGE

COURSE 3 · CH 12 · FINAL CHAPTER · CAPSTONE

Capstone — A Multi-Feature Page

Combining web components, canvas, lazy-loaded media, accessibility patterns, and internationalization into one realistic page

This final chapter of the entire HTML series doesn't introduce anything new — it builds one page genuinely combining everything from this Advanced course, on top of everything from Fundamentals and Intermediate before it.

The Page

```
<!DOCTYPE html>
<html lang="en"> // Advanced Ch11
<head>
  <meta charset="UTF-8"> // must be early — Ch11
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Project Showcase</title>
  <link rel="manifest" href="/manifest.json"> // Ch7
  <link rel="preload" href="/fonts/main.woff2" as="font" crossorigin> // Ch9
</head>
<body>

  <header>
    <h1>Project Showcase</h1>
  </header>

  <main>
    // Accessible tabs — Ch8's full ARIA pattern with roving tabindex
    <div role="tablist" aria-label="Project views">
      <button role="tab" aria-selected="true" aria-controls="panel-canvas">Live Drawing</button
      <button role="tab" aria-selected="false" aria-controls="panel-gallery" tabindex="-1">Gall
    </div>

    <div role="tabpanel" id="panel-canvas">
      // Canvas — Ch2's animation loop
      <canvas id="drawing" width="400" height="200"></canvas>
    </div>
```

```
    <div role="tabpanel" id="panel-gallery" hidden>
      // Lazy-loaded images — Ch9, never on the visible-by-default tab
      <img src="project1.jpg" loading="lazy" alt="Screenshot of project 1">
    </div>

    // Web component — Ch1's custom element with shadow DOM and a real slot
    <rating-stars value="4">
      <span slot="label">Reader rating</span>
    </rating-stars>

    // Foreign-language phrase — Ch11's inline lang attribute
    <p>Built with <span lang="fr">savoir-faire</span>.</p>
  </main>

  <script src="app.js" defer></script> // Ch9 — defer, not blocking

</body>
</html>
```

What Each Decision Connects Back To

- **Tabs with roving tabindex** — Chapter 8's full ARIA pattern, not a native element, since custom panel content (canvas, gallery) genuinely needs this level of control.
- **Canvas only inside the active tab's panel** — no point running an animation loop the user can't see; pairs naturally with starting/stopping the `requestAnimationFrame` loop based on which tab is active.
- **loading="lazy" on gallery images** — Chapter 9's rule applied correctly: these are in a hidden panel, genuinely not needed until the user switches tabs.
- **<rating-stars> web component with a named slot** — Chapter 1's pattern, letting the page supply its own label text while the component handles the star rendering internally.
- **defer on the script** — Chapter 9, ensures the DOM (including the custom element registration) is ready before app.js runs.

Final Series-Wide Checklist

- [x] **Valid, semantic document structure throughout**
  Doctype, lang, semantic landmarks, no div soup where a real element would fit.
  Fundamentals Ch1, 8, 9

- [x] **Every interactive widget is keyboard-operable and correctly announced**
  Roving tabindex on tabs, real button/label elements, no click-only divs.
  Intermediate Ch6, Advanced Ch8

- [x] **Off-screen/hidden content is lazy-loaded; nothing above-the-fold is delayed**
  Advanced Ch9

- [x] **Validated against the W3C validator before considering it done**
  Fundamentals Ch10

- [x] **Tested with an actual screen reader, not just automated tools**
  Advanced Ch8

THIS IS THE SAME LESSON THE SERIES HAS RETURNED TO REPEATEDLY: MEANING BEFORE APPEARANCE

Every chapter across all three courses, in one way or another, came back to the same underlying idea — choose the element, attribute, or pattern that genuinely describes what something *is* and what it *does*, and the browser, search engines, and assistive technology all reward that choice automatically. Appearance is CSS's job; HTML's job, done well, is meaning.

CHAPTER 12 QUICK REFERENCE — AND SERIES WRAP-UP

- **Course 3 recap:** web components in depth (Ch1) → canvas (Ch2) → SVG (Ch3) → drag-and-drop (Ch4) → File API (Ch5) → storage/offline (Ch6) → PWAs (Ch7) → accessibility deep dive (Ch8) → performance (Ch9) → APIs grab-bag (Ch10) → internationalization (Ch11) → capstone (Ch12)
- **Full HTML series complete:** 36 chapters across Fundamentals, Intermediate, and Advanced
- **The series-wide thread:** semantic meaning first — every advanced technique still rests on Fundamentals' core principle

- **What's next for this content area:** the CSS courses (already complete) cover appearance; a JavaScript-focused course would be the natural remaining gap if Philip wants to continue this line further