HTML SERIES · COURSE 2

HTML Intermediate

Forms, tables, multimedia, iframes, meta tags, accessibility, structured data, web components, and a real project build. 12 complete chapters.

12 Chapters

osztromok.com

CHAPTER 1: FORMS IN DEPTH

COURSE 2 · CH 1

Forms in Depth

The full range of input types, built-in validation attributes, and fieldset/legend for grouping related fields

Fundamentals Course Chapter 7 covered the basic shape of a form. This chapter covers what makes modern HTML forms genuinely powerful: a wide range of specialised input types, validation that works without any JavaScript at all, and proper grouping for related fields.

The Full Range of Input Types

`date` A native date picker	**`time`** A native time picker	**`datetime-local`** Combined date and time picker
`tel` Phone number — triggers the numeric keypad on mobile	**`url`** Web address, with basic URL-format validation	**`search`** Like text, but often styled with a clear/cancel button by the browser
`color` A native colour picker, returns a hex value	**`range`** A slider, between a min and max value	**`file`** Upload a file — full coverage in Course 3's File API chapter
`hidden` Not rendered at all, but still submitted with the form	**`textarea`** Multi-line text — a separate tag, not an input type	**`select`** A dropdown — also a separate tag, with nested option elements

```
<input type="date" name="appointment">
<input type="range" name="volume" min="0" max="100">

<textarea name="message" rows="4"></textarea>

<select name="country">
  <option value="uk">United Kingdom</option>
  <option value="hu">Hungary</option>
</select>
```

Built-In Validation — No JavaScript Required

The browser itself can enforce a meaningful set of validation rules, blocking form submission and showing a native error message until they're satisfied — all without a single line of JavaScript.

ATTRIBUTE	WHAT IT ENFORCES
`required`	The field must have a value before the form can submit
`minlength` / `maxlength`	Text length boundaries
`min` / `max`	Numeric or date value boundaries
`pattern`	A regular expression the value must match — full regex syntax, same engine as the Regex course covered elsewhere
`step`	Increment granularity for number/range inputs

```
<input type="text" name="username" required minlength="3" maxlength="20">

// pattern with a regex — UK-style postcode, simplified
<input type="text" name="postcode" pattern="[A-Z]{1,2}[0-9][A-Z0-9]? ?[0-9][A-Z]{2}">
```

BUILT-IN VALIDATION IS A UX CONVENIENCE, NOT A SECURITY BOUNDARY

Native HTML validation runs entirely in the browser and can be bypassed trivially — disabled JavaScript-free, a direct API request, or simply editing the HTML in DevTools all skip it completely. **Server-side validation of every submitted value is always required** regardless of what client-side validation exists — this is the same underlying principle as the browser security course's trust boundary discussions.

fieldset and legend — Grouping Related Fields

For a form with logically related groups of fields (a shipping address section, a set of radio button options), `<fieldset>` provides both a visual grouping (a default border) and a genuine semantic one, with `<legend>` as its caption.

```
<fieldset>
  <legend>Shipping Address</legend>

  <label for="street">Street</label>
  <input type="text" id="street" name="street">
```

```
  <label for="city">City</label>
  <input type="text" id="city" name="city">
</fieldset>
```

FIELDSET IS ESPECIALLY VALUABLE FOR A GROUP OF RADIO BUTTONS

A screen reader announces the legend text alongside each radio button in the group, giving full context ("Shipping speed: Standard," "Shipping speed: Express") rather than just the bare option label alone — genuinely improves the experience for assistive technology users navigating between related options.

Disabling Part of a Form — fieldset disabled

```
<fieldset disabled>
  // Every input inside is disabled in one step, without setting it on each individually
</fieldset>
```

CHAPTER 1 QUICK REFERENCE

- **Specialised input types:** date, time, tel, url, color, range, file, hidden — each with a tailored native UI
- **textarea/select** — separate tags, not input types, for multi-line text and dropdowns
- **required/minlength/maxlength/min/max/pattern/step** — built-in validation, no JavaScript needed
- **Client-side validation is convenience, not security** — always re-validate every value server-side too
- **fieldset/legend** — semantic grouping with a real accessibility benefit, especially for radio button groups
- **fieldset disabled** — disables every contained input in one step
- **Next chapter:** tables in depth — colspan/rowspan, complex layouts, full accessibility (scope, caption)

CHAPTER 2: TABLES IN DEPTH

COURSE 2 · CH 2

Tables in Depth

colspan/rowspan in real layouts, and the full accessibility picture beyond Fundamentals Chapter 6's basics

Fundamentals Chapter 6 introduced `scope`, `caption`, and basic spanning. This chapter goes further — building a genuinely complex, multi-level header table, and covering the accessibility attributes that matter once a table's structure is no longer simple rows and columns.

A Complex Header Structure with colspan/rowspan

```
<table>
  <tr>
    <th rowspan="2">Product</th>
    <th colspan="2">2026 Sales</th>
  </tr>
  <tr>
    <th>Q1</th>
    <th>Q2</th>
  </tr>
  <tr>
    <td>Widget</td>
    <td>£420</td>
    <td>£510</td>
  </tr>
</table>
```

Product	2026 Sales	
	Q1	Q2
Widget	£420	£510

"Product" spans two rows since it has no separate sub-category; "2026 Sales" spans two columns since it's a grouping header for Q1 and Q2 beneath it. Working out colspan/rowspan values is genuinely easier by sketching the grid on paper first — getting them wrong silently shifts every following cell out of alignment.

A SINGLE MISCOUNTED COLSPAN/ROWSPAN SILENTLY MISALIGNS THE ENTIRE REST OF THE TABLE

Unlike most HTML mistakes, a wrong span value doesn't produce an error — it just shifts subsequent cells, sometimes subtly, sometimes dramatically. Counting carefully (or building from a sketched grid) before writing the markup avoids a genuinely tedious class of bug to debug after the fact.

headers Attribute — Explicit Cell-to-Header Association

Fundamentals Chapter 6's `scope` handles simple row/column relationships well. For a genuinely complex table — like the multi-level header above — a data cell can explicitly reference *which specific header IDs* describe it, for the most precise possible accessibility association.

```
<th id="product">Product</th>
<th id="q1">Q1</th>
// ...
<td headers="product q1">£420</td>
```

Reserve `headers` for cases where `scope` alone genuinely can't express the relationship — a cell that depends on headers from both a row group AND a column group simultaneously, exactly the multi-level case above.

Sortable Table Headers — A Common Real-World Pattern

Many real tables let a visitor click a header to sort by that column — purely a JavaScript behaviour, but the underlying markup convention is worth knowing.

```
<th>
  <button type="button">Name <span>↕</span></button>
</th>
```

Using a genuine `<button>` inside the header cell, rather than attaching a click handler directly to the `<th>`, keeps the element keyboard-accessible and properly announced as interactive — exactly the kind of distinction the browser security/accessibility courses return to repeatedly.

Responsive Tables — A Brief Practical Note

Wide tables on narrow (mobile) screens are a genuinely hard layout problem, solved primarily with CSS rather than HTML markup changes — wrapping the table in a horizontally scrollable container is the most common, robust approach:

```
<div style="overflow-x: auto;">
  <table>...</table>
</div>
```

Full coverage of responsive layout techniques belongs in the CSS courses — this is included here purely because it's such a common real-world pairing with genuinely wide tables.

CHAPTER 2 QUICK REFERENCE

- **colspan/rowspan** in multi-level headers — sketch the grid first to avoid silent misalignment
- **headers="id1 id2"** on a <td> — explicit header association for cases scope alone can't express
- **Sortable headers:** wrap the clickable label in a real <button>, not a click handler on <th> directly
- **Wide tables on mobile** — wrap in a horizontally scrollable container; full responsive technique belongs in CSS
- **Next chapter:** multimedia in depth — video/audio attributes, tracks/subtitles, the picture element for responsive images

CHAPTER 3: MULTIMEDIA IN DEPTH

COURSE 2 · CH 3

Multimedia in Depth

Video/audio attributes beyond the basics, captions and subtitles, and the picture element for genuinely responsive images

Fundamentals Chapter 5 covered basic image and media embedding. This chapter goes further into making media genuinely accessible (captions, subtitles) and genuinely responsive — serving different image files for different screen sizes or pixel densities, not just resizing one image with CSS.

The track Element — Captions and Subtitles

```
<video controls>
  <source src="tutorial.mp4" type="video/mp4">
  <track kind="subtitles" src="subtitles-en.vtt" srclang="en" label="English" default>
  <track kind="subtitles" src="subtitles-hu.vtt" srclang="hu" label="Hungarian">
</video>
```

KIND VALUE	PURPOSE
`subtitles`	Translated dialogue, for viewers who don't understand the spoken language
`captions`	Dialogue AND significant sound effects, for deaf/hard-of-hearing viewers — same language as the audio
`descriptions`	Text describing visual content, for blind/visually-impaired viewers
`chapters`	Navigation points within the video, for chapter-jump UI

The actual subtitle/caption text lives in a separate `.vtt` (WebVTT) file — a simple timestamped text format, not embedded directly in the HTML itself.

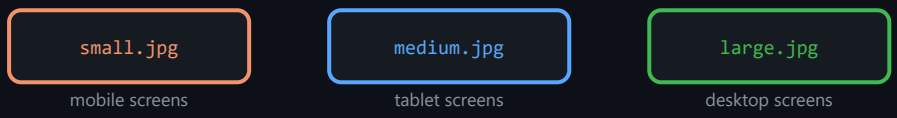
SUBTITLES AND CAPTIONS ARE NOT THE SAME THING, DESPITE OFTEN BEING USED INTERCHANGEABLY CASUALLY

Subtitles assume the viewer can hear the audio but not understand the language — they translate dialogue only. Captions assume the viewer cannot hear the audio at all — they include both dialogue AND

meaningful sound effects ("[door slams]," "[tense music playing]"). Choosing the right `kind` value, and writing the actual content with the right audience in mind, genuinely matters for accessibility.

The picture Element — Responsive Images

A single `<img>` always loads the same file, regardless of screen size — wasteful on a small mobile screen if the only available image is a large desktop-resolution photo. `<picture>` lets the browser choose between several source files based on screen width or pixel density.



The browser picks ONE based on the matching media condition

Only the chosen image actually downloads — not all three, saving real bandwidth

```
<picture>
  <source media="(max-width: 600px)" srcset="small.jpg">
  <source media="(max-width: 1200px)" srcset="medium.jpg">
  <img src="large.jpg" alt="A description of the image">
</picture>
```

Critically, the `<img>` tag is still required inside `<picture>` — it's the fallback for any browser that doesn't support `<picture>` at all, and it's also where `alt` text (Fundamentals Chapter 5) belongs, regardless of which `<source>` ends up actually used.

srcset on a Plain img — Pixel Density Variants

A simpler, related technique: offering multiple resolutions of the same image for different screen pixel densities (a high-DPI "retina" display vs a standard one), without needing the full `<picture>` structure.

```
<img src="photo.jpg"
     srcset="photo.jpg 1x, photo@2x.jpg 2x, photo@3x.jpg 3x"
     alt="...">
```

PICTURE IS FOR ART DIRECTION; SRCSET ALONE IS FOR RESOLUTION SWITCHING

Use `<picture>` when genuinely *different images* (a cropped close-up on mobile vs a wide shot on desktop) are needed for different contexts. Use plain `<img srcset>` when it's the *same image*, just at different resolutions for different screen pixel densities — a meaningfully different problem each technique is purpose-built to solve.

preload — Hinting How Eagerly to Load Media

```
<video controls preload="metadata"> // loads just duration/dimensions, not the full video
<video controls preload="none"> // loads nothing until the user presses play
```

CHAPTER 3 QUICK REFERENCE

- **<track kind="subtitles/captions/descriptions/chapters">** — references a separate .vtt timestamped text file
- **Subtitles ≠ captions** — subtitles translate dialogue only; captions include meaningful sound effects too
- **<picture>** — browser picks ONE source based on matching media conditions; always include a fallback <img> with alt
- **picture = art direction** (different images); **srcset alone = resolution switching** (same image, different pixel densities)
- **preload="metadata"/"none"** — controls how eagerly media loads before playback starts
- **Next chapter:** iframes & embeds

CHAPTER 4: IFRAMES & EMBEDS

COURSE 2 · CH 4

iframes & Embeds

Embedding another page within yours — what it's good for, and the real security implications worth understanding

An `<iframe>` embeds an entirely separate HTML document within the current page — a YouTube video, a Google Map, a payment widget. It's one of the few HTML elements with genuine, non-trivial security implications worth understanding properly, directly connecting to the browser security course's clickjacking and CORS coverage.

Basic Usage

```
<iframe src="https://www.youtube.com/embed/VIDEO_ID"
        width="560" height="315"
        title="Video title for accessibility"
        allowfullscreen>
</iframe>
```

TITLE IS REQUIRED FOR ACCESSIBILITY, EASILY FORGOTTEN

Without a `title` attribute, a screen reader has no meaningful way to describe what an embedded iframe actually contains to the user — exactly the same accessibility gap covered for images (alt text, Fundamentals Chapter 5) and links (descriptive text, Fundamentals Chapter 4), applied here to embedded content instead.

Common Real-World Uses

📺 Video embeds

YouTube, Vimeo — the standard way these platforms provide embeddable players without needing to host video files yourself.

🗺️ Maps

Google Maps and similar services provide an iframe embed code for showing an interactive map without a full API integration.

💳 Payment widgets

Many payment processors deliberately use an iframe so your page never directly touches card details — the sensitive form lives entirely within the processor's own, separately-secured origin.

📊 Embedded dashboards

Analytics tools, survey forms, and similar third-party widgets commonly provide an iframe snippet as their integration method.

The Security Side — Why iframes Are Treated Carefully

Because an iframe loads a genuinely separate document, the Same-Origin Policy (covered fully in the browser security course's CORS chapter) governs what the parent page and the embedded page can and can't do to each other — by default, scripts in each can't reach into the other's content at all if they're on different origins.

EMBEDDING UNTRUSTED CONTENT IS EXACTLY THE CLICKJACKING RISK COVERED IN THE BROWSER SECURITY COURSE

A malicious page could iframe your site invisibly and trick a logged-in visitor into clicking something on your site without realising it. This is why `X-Frame-Options` / `frame-ancestors` (covered fully in the browser security course) exist on the receiving end — and why embedding genuinely untrusted third-party content in your own iframe deserves the same caution in reverse.

The sandbox Attribute — Restricting an Embedded Page's Capabilities

For embedding content you don't fully trust, `sandbox` lets you strip away specific capabilities from the embedded document — by default, an empty `sandbox` attribute restricts almost everything; specific values selectively re-enable just what's needed.

```
// Maximally restrictive — almost everything disabled
<iframe src="untrusted-content.html" sandbox></iframe>

// Selectively re-enable scripts and form submission only
<iframe src="widget.html" sandbox="allow-scripts allow-forms"></iframe>
```

VALUE	RE-ENABLES
`allow-scripts`	JavaScript execution within the iframe

VALUE	RE-ENABLES
`allow-forms`	Form submission from within the iframe
`allow-popups`	Opening new windows/tabs from within the iframe
`allow-same-origin`	Treats the iframe as same-origin if it genuinely is — use carefully, as this can reintroduce risks the sandbox otherwise prevents

START WITH THE MOST RESTRICTIVE SANDBOX AND ADD ONLY WHAT'S GENUINELY NEEDED

Adding `sandbox` with no values at all is the safest starting point — then adding specific `allow-*` values one at a time, only as something is confirmed to actually break without it, keeps the embedded content's capabilities as minimal as the use case genuinely requires.

loading="lazy" — Deferring Off-Screen iframes

```
<iframe src="..." loading="lazy" title="..."></iframe>
```

Defers loading an iframe until it's about to scroll into view — genuinely useful for a page with several embedded widgets further down the page that most visitors never scroll to, directly improving initial page load performance.

CHAPTER 4 QUICK REFERENCE

- **<iframe src="..." title="...">** — embeds a separate document; title is required for accessibility
- **Common uses:** video embeds, maps, payment widgets (deliberately isolated origin), embedded dashboards
- **Same-Origin Policy** governs cross-origin iframe interaction — covered fully in the browser security course
- **sandbox** — restricts an embedded page's capabilities; start maximally restrictive, add allow-* only as genuinely needed
- **allow-same-origin** can reintroduce risk — use carefully, only when the embedded content genuinely is same-origin
- **loading="lazy"** — defers off-screen iframes, improving initial page load

- **Next chapter:** meta tags — SEO basics, viewport, Open Graph & Twitter card social sharing tags

CHAPTER 5: META TAGS

COURSE 2 · CH 5

Meta Tags

SEO basics, the viewport tag every responsive site needs, and the Open Graph/Twitter Card tags that control how links look when shared

Everything in this chapter lives inside `<head>` (Fundamentals Chapter 1) and is invisible on the page itself — but each tag here has real, visible consequences in search results, social media previews, and mobile rendering.

The Essential SEO Meta Tags

```
<title>Linux Performance Tuning — Free Course</title>
<meta name="description" content="Learn to diagnose CPU, memory, disk, and network bottleneck
```

Linux Performance Tuning — Free Course

osztromok.com › linux › performance

Learn to diagnose CPU, memory, disk, and network bottlenecks on Linux, scenario by scenario.

`<title>` (already covered in Fundamentals Chapter 1) is what search engines typically show as the clickable headline; the `description` meta tag is typically shown as the snippet text beneath it — though search engines sometimes substitute their own extracted text if they judge it more relevant to a specific search.

A UNIQUE TITLE AND DESCRIPTION PER PAGE MATTERS MORE THAN PEOPLE EXPECT

Using the exact same title and description across every page of a site (a common mistake on small sites) makes search results indistinguishable from each other, and search engines themselves weigh page-specific, unique titles more favourably than identical, generic ones repeated site-wide.

The viewport Tag — Required for Any Responsive Site

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

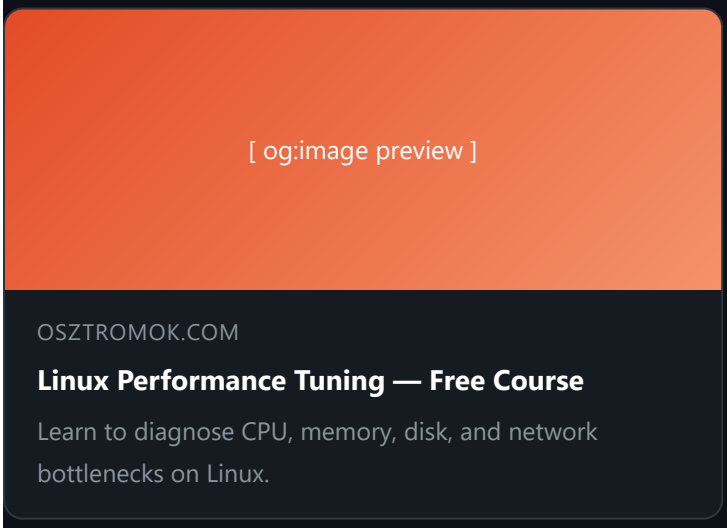
Without this tag, mobile browsers render the page at a fixed desktop-sized virtual width and zoom out to fit it on screen — every bit of CSS responsive design relies on this tag being present to actually work as intended on a real phone. `width=device-width` matches the page's width to the actual device; `initial-scale=1` sets the starting zoom level to 100%.

A MISSING VIEWPORT TAG IS ONE OF THE MOST COMMON REASONS A "RESPONSIVE" SITE DOESN'T ACTUALLY LOOK RESPONSIVE ON A PHONE

The CSS media queries (covered in the CSS courses) that make a layout adapt to screen size depend entirely on the browser knowing the real device width — without this single meta tag, all of that responsive CSS effectively does nothing on mobile.

Open Graph — Controlling Facebook/LinkedIn Link Previews

```
<meta property="og:title" content="Linux Performance Tuning — Free Course">
<meta property="og:description" content="Learn to diagnose CPU, memory, disk, and network bot
<meta property="og:image" content="https://osztromok.com/images/linux-course-card.jpg">
<meta property="og:url" content="https://osztromok.com/linux/performance">
```



Without these tags, a link shared on Facebook, LinkedIn, or Discord often shows a generic, unhelpful preview — or guesses at an image/description from the raw page content, sometimes incorrectly. Open Graph tags give explicit control over exactly what appears.

Twitter Cards — A Parallel System for X/Twitter

```
<meta name="twitter:card" content="summary_large_image">
<meta name="twitter:title" content="Linux Performance Tuning — Free Course">
```

```
<meta name="twitter:description" content="Learn to diagnose CPU, memory, disk, and network bo
<meta name="twitter:image" content="https://osztromok.com/images/linux-course-card.jpg">
```

X/TWITTER FALLS BACK TO OPEN GRAPH TAGS IF TWITTER-SPECIFIC ONES ARE MISSING

Adding both sets of tags is the most thorough approach, but in practice many sites get away with just Open Graph tags plus a single `twitter:card` value, since X's crawler checks for Open Graph equivalents when its own specific tags aren't present.

Quick Reference of Common Meta Tags

TAG	PURPOSE
`meta charset="UTF-8"`	Character encoding — should be the very first thing inside <head>
`meta name="description"`	Search engine snippet text
`meta name="viewport"`	Required for any responsive layout to work on mobile
`meta name="robots" content="noindex"`	Tells search engines NOT to index this specific page
`link rel="canonical"`	Declares the "official" URL when the same content is reachable via multiple URLs

CHAPTER 5 QUICK REFERENCE

- **title + meta description** — what typically shows in search results; keep both unique per page
- **meta viewport** — required for responsive CSS to actually function on mobile devices
- **Open Graph (og:title/description/image/url)** — controls Facebook/LinkedIn/Discord link previews
- **Twitter Cards (twitter:card/title/description/image)** — parallel system for X/Twitter, falls back to Open Graph if missing
- **meta charset** — should be the very first tag inside <head>
- **meta robots / link canonical** — fine-grained control over search engine indexing behaviour
- **Next chapter:** accessibility (a11y) fundamentals — ARIA roles, landmarks, alt text best practices

CHAPTER 6: ACCESSIBILITY FUNDAMENTALS

COURSE 2 · CH 6

Accessibility (a11y) Fundamentals

ARIA roles, landmarks, and a deliberate consolidation of the alt-text/accessibility habits introduced piece by piece across Course 1

Accessibility hasn't been a separate topic in this series — it's been woven into nearly every chapter so far (alt text in Ch5, label/for in Ch7, scope in tables Ch6, semantic landmarks in Ch8). This chapter consolidates that into a deliberate, named topic, and introduces ARIA — the toolkit for the cases semantic HTML alone doesn't fully cover.

The First Rule of ARIA: Don't Use ARIA If You Don't Need To

This sounds counterintuitive in a chapter about ARIA, but it's the single most important guideline in the whole topic: a native HTML element with correct built-in semantics is almost always better than a generic element with ARIA attributes bolted on.

⚠️ More work, more risk

`<div role="button" tabindex="0" onclick="...">` — needs manual keyboard handling, manual focus styling, manual Enter/Space key support, and is easy to get subtly wrong.

✅ Correct by default

`<button>` — keyboard accessibility, focus handling, and correct screen reader announcement all come built in, for free, automatically.

ARIA Landmark Roles — Reinforcing Semantic HTML

Fundamentals Chapter 8's semantic elements (header, nav, main, footer) already carry implicit ARIA landmark roles automatically — this is exactly why using real semantic tags is preferable to adding roles manually onto generic divs.

```
// Already has an implicit role of "navigation" — no extra ARIA needed
<nav>...</nav>
```

```
// Only needed when a genuinely non-semantic element must serve a landmark role
<div role="navigation">...</div>
```

ARIA Attributes That Genuinely Add Value

aria-label

Provides an accessible name when visible text alone isn't descriptive enough — e.g. an icon-only close button.

aria-describedby

Points to another element's id whose text serves as additional description — useful for linking a form field to detailed help text beyond its label.

aria-hidden="true"

Hides a purely decorative element from screen readers entirely — for icons that add no information beyond what's already conveyed in text nearby.

aria-expanded

Communicates whether a collapsible element (an accordion, a dropdown menu) is currently open or closed — toggled via JavaScript as the state changes.

aria-live="polite"

Announces dynamically updated content (a "3 new messages" counter, a form validation message appearing) without interrupting whatever the screen reader is currently reading.

```
// Icon-only button — aria-label provides the accessible name
<button aria-label="Close dialog">✕</button>

// Decorative icon next to text that already says the same thing
<button>
  <svg aria-hidden="true">...</svg>
  Delete item
</button>
```

Revisiting alt Text — Practical Guidance, Consolidated

Fundamentals Chapter 5 introduced the basics; here's the consolidated practical guidance worth internalising properly:

- **Describe content and function, not appearance.** "Submit form" not "blue rectangular button," unless the visual styling itself is genuinely the relevant information.
- **Don't repeat information already in adjacent text.** If a caption already says "Philip's Raspberry Pi setup," the image's alt doesn't need to repeat it verbatim.
- **For complex images (charts, diagrams), summarise the key takeaway** rather than attempting to describe every visual detail exhaustively — "Bar chart showing 23% growth in Q3" beats a literal pixel-by-pixel description.
- **For purely decorative images, alt=""** (Fundamentals Chapter 5's rule, repeated because it matters) — not a missing attribute, an explicit empty one.

Keyboard Accessibility — The Test Most Sites Fail Silently

A genuinely accessible page can be fully operated using only the Tab, Enter, and Space keys — no mouse at all. The fastest practical check: unplug the mouse (or just don't touch it) and try to use the entire page with the keyboard alone.

A VISIBLE FOCUS INDICATOR MUST NEVER BE REMOVED WITHOUT A GENUINELY GOOD REPLACEMENT

`* { outline: none; }` in CSS (sometimes added purely to "clean up" the visual look) removes the only visible signal of which element currently has keyboard focus — leaving keyboard users with no way to tell where they are on the page at all. If the default focus outline's appearance is genuinely undesirable, replace it with a custom, clearly visible style — never simply remove it.

Testing Accessibility in Practice

- **Browser DevTools' Accessibility panel** shows the computed accessible name, role, and properties for any selected element — directly verifies what a screen reader would actually announce.
- **Automated checkers** (Lighthouse, axe) catch a meaningful subset of issues automatically — missing alt text, insufficient colour contrast, missing form labels.
- **Manual keyboard-only testing** catches what automated tools generally can't — genuinely logical tab order, working focus management in custom widgets.

AUTOMATED TOOLS CATCH MAYBE A THIRD OF REAL ACCESSIBILITY ISSUES — MANUAL TESTING STILL MATTERS

Lighthouse and axe are genuinely valuable first-pass tools, but they can only catch programmatically detectable issues (a missing attribute, insufficient contrast) — they cannot judge whether alt text is actually meaningful, or whether a custom widget's keyboard behaviour genuinely makes sense to use. Treat automated checks as a floor, not a ceiling.

CHAPTER 6 QUICK REFERENCE

- **First rule of ARIA:** a correct native element beats ARIA-on-a-div almost every time
- **Semantic HTML landmarks** (Fundamentals Ch8) already carry implicit ARIA roles — no extra work needed
- **aria-label/aria-describedby/aria-hidden/aria-expanded/aria-live** — genuinely valuable when native HTML can't express the needed meaning alone
- **alt text:** describe content/function not appearance, don't repeat adjacent text, summarise complex images, empty alt for decorative ones
- **Keyboard test:** can the entire page be operated with Tab/Enter/Space alone?
- **Never remove the focus outline without a clear custom replacement**
- **Automated tools (Lighthouse/axe) catch a floor of issues**, not the full picture — manual testing still matters
- **Next chapter:** data attributes & custom attributes

CHAPTER 7: DATA ATTRIBUTES & CUSTOM ATTRIBUTES

COURSE 2 · CH 7

Data Attributes & Custom Attributes

A standards-compliant way to attach your own data to an HTML element, for CSS and JavaScript to read

Sometimes an element genuinely needs to carry information that has no existing HTML attribute to hold it — a product ID, a sort key, a state flag. `data-*` attributes are the official, validator-approved way to do exactly this, without inventing non-standard attributes that would fail validation (Fundamentals Chapter 10).

The data-* Syntax

```
<li data-product-id="4821" data-category="electronics">
  Wireless Keyboard
</li>
```

Any attribute name starting with `data-` is reserved specifically for this purpose and guaranteed never to clash with a current or future standard HTML attribute — exactly why the prefix exists, rather than just inventing arbitrary attribute names directly.

Reading data-* Attributes in JavaScript

```
// HTML
<li data-product-id="4821">Wireless Keyboard</li>

// JavaScript — the dataset API converts data-product-id automatically
const item = document.querySelector('li');
console.log(item.dataset.productId); // "4821"
```

Note the naming conversion: a hyphenated attribute name (`data-product-id`) becomes camelCase in JavaScript's `dataset` object (`productId`) — this conversion happens automatically and consistently, the same convention used across the DOM API generally.

Reading data-* Attributes in CSS

```
// HTML
<div data-status="urgent">Task item</div>

// CSS — attribute selector, no JavaScript needed at all
[data-status="urgent"] {
  border-left: 4px solid red;
}
```

CSS can select and style elements based on a data attribute's value directly, using a standard attribute selector — genuinely useful for visually distinguishing states (urgent/normal/done, active/inactive) without any scripting at all.

Common Real-World Uses

🏷️ Storing an ID for JS to use later

A list item carrying a database record's ID, read when the user clicks it — avoiding needing a separate lookup table mapping DOM elements to IDs.

🎯 Sort/filter keys

A data-price or data-date attribute holding a raw sortable value, separate from the formatted, human-readable text actually displayed.

🔄 State flags

data-expanded="true" on a collapsible section — readable by both CSS (styling) and JS (behaviour) from the same single source of truth.

🧪 Test hooks

data-testid="submit-button" — a stable identifier for automated testing tools, deliberately decoupled from CSS classes that might change for styling reasons.

Sort Key Example — Separating Display Text from Sortable Value

```
<ul>
  <li data-price="9.99">£9.99</li>
  <li data-price="149.00">£149.00</li>
</ul>
```

Sorting by the displayed text "£9.99" vs "£149.00" alphabetically would put £149.00 first — wrong. Sorting by the raw numeric `data-price` value instead gives the genuinely correct order. This pattern — display text for humans, a clean machine-readable value in a data attribute — comes up constantly in real interfaces.

DATA-* ATTRIBUTES ARE VISIBLE IN THE PAGE SOURCE — NEVER STORE SECRETS IN THEM

Exactly the same principle as HTML comments (Fundamentals Chapter 10): anything in a data attribute is plainly visible to anyone viewing the page source. Fine for a product ID or a sort key; never appropriate for anything sensitive.

Why Not Just Invent Your Own Attribute Name?

Writing `<li productid="4821">` directly (without the `data-` prefix) works in most browsers today, but is technically invalid HTML — and risks a genuine future collision if HTML's specification ever adds a real, standard attribute with that exact name. The `data-*` prefix is reserved specifically so this can never happen — a small habit that costs nothing and avoids a real, if rare, category of future bug.

CHAPTER 7 QUICK REFERENCE

- **data-*** — the only standards-compliant way to attach custom data to an HTML element
- **JavaScript:** element.dataset.propertyName — hyphenated attribute names convert to camelCase automatically
- **CSS:** [data-attribute="value"] — a standard attribute selector, no JavaScript needed
- **Common uses:** storing IDs for later JS use, sort/filter keys, state flags, stable test hooks (data-testid)
- **Sort key pattern:** keep a clean numeric value in a data attribute, separate from formatted display text
- **Never store secrets** in data attributes — visible in page source like any other attribute
- **Always use the data- prefix** rather than inventing arbitrary attribute names — avoids future spec collisions, keeps the page valid
- **Next chapter:** HTML APIs overview — template element, details/summary, dialog element

CHAPTER 8: HTML APIS OVERVIEW

COURSE 2 · CH 8

HTML APIs Overview

template, details/summary, and dialog — modern elements that replace JavaScript-heavy patterns with native browser behaviour

Modern HTML has steadily absorbed UI patterns that used to require significant JavaScript — collapsible sections, modal dialogs, reusable markup fragments. This chapter covers three elements that genuinely reduce how much custom code a typical interactive page needs.

details and summary — Native Collapsible Content

```
<details>
  <summary>What's included in the course?</summary>
  <p>12 chapters covering everything from basics to a real project build.</p>
</details>
```

▶ What's included in the course?

▶ Is it free?

No JavaScript at all — click the `<summary>` text and the rest of the content toggles visible/hidden, with built-in keyboard support (Enter/Space) and correct screen reader announcement of the expanded/collapsed state, directly connecting to Chapter 6's `aria-expanded` coverage — except here, the browser itself manages that state automatically.

```
// Open by default, instead of collapsed
<details open>
  <summary>...</summary>
</details>
```

A CLASSIC, REAL-WORLD USE: AN ENTIRELY JAVASCRIPT-FREE FAQ ACCORDION

A series of `<details>` elements, each with its own question in `<summary>` and the answer beneath, produces a fully functional, accessible FAQ accordion with zero JavaScript — exactly the kind of interactive pattern this chapter's elements exist to simplify.

The dialog Element — Native Modal Dialogs

```
<dialog id="confirmDialog">
  <p>Are you sure you want to delete this item?</p>
  <button onclick="document.getElementById('confirmDialog').close()">Cancel</button>
</dialog>

<button onclick="document.getElementById('confirmDialog').showModal()">Delete</button>
```

`<dialog>` is hidden by default; calling `.showModal()` via JavaScript displays it as a genuine modal — automatically including a semi-transparent backdrop, trapping keyboard focus inside the dialog (so Tab can't accidentally reach content behind it), and closing on the Escape key, all built into the browser.

BEFORE DIALOG EXISTED, "MODAL" WAS ALMOST ALWAYS A HAND-BUILT DIV WITH SIGNIFICANT JAVASCRIPT

Focus trapping, backdrop click handling, Escape-key closing, and correct ARIA roles all had to be implemented manually for a custom modal — and were genuinely easy to get subtly wrong, especially for keyboard/screen reader users. `<dialog>` handles the correct behaviour by default, which is exactly why it's worth preferring over a hand-rolled equivalent whenever a genuine modal is needed.

showModal() vs show()

`.showModal()`

True modal behaviour — backdrop, focus trapping, blocks interaction with the rest of the page until closed.

`.show()`

Non-modal display — no backdrop, no focus trapping, the rest of the page remains fully interactive. Rarely what's actually wanted for a typical "modal" use case.

The template Element — Inert, Reusable Markup

```
<template id="productCardTemplate">
  <div class="product-card">
    <h3></h3>
    <p class="price"></p>
  </div>
</template>

// JavaScript clones the template's content as needed:
const tpl = document.getElementById('productCardTemplate');
const clone = tpl.content.cloneNode(true);
// ...fill in the clone's content, then append it to the page
```

Content inside `<template>` is parsed by the browser but never rendered or executed (images don't load, scripts don't run) until explicitly cloned and inserted into the document via JavaScript — a genuinely useful pattern for repeated UI structures (a card layout, a list item) built once and stamped out multiple times with different data.

Comparing the Pre-Built-By-The-Browser Approach

- **Without template:** building the same HTML structure repeatedly via string concatenation or `innerHTML` assignments in JavaScript — error-prone, and risks the same kind of escaping mistakes covered in Fundamentals Chapter 11's XSS warning if any of the inserted data comes from user input.
- **With template:** the structure lives as real, validatable HTML; JavaScript just clones it and fills in the blanks — generally safer and easier to maintain.

CHAPTER 8 QUICK REFERENCE

- **details/summary** — native, JavaScript-free collapsible content with built-in keyboard and accessibility support
- **details open** — starts expanded instead of collapsed
- **dialog + showModal()** — native modal: backdrop, focus trapping, Escape-to-close, all built in
- **show() vs showModal()** — show() has no backdrop/focus trapping, rarely what's wanted for a true modal
- **template** — inert markup, never rendered until cloned via JavaScript; safer/cleaner than building HTML via string concatenation
- **All three reduce custom JavaScript** for patterns that used to require it entirely

- **Next chapter:** microdata & structured data (schema.org) for SEO

CHAPTER 9: MICRODATA & STRUCTURED DATA

COURSE 2 · CH 9

Microdata & Structured Data

Schema.org vocabulary, embedded directly in your markup, that earns rich search result features other pages don't get

Search engines parse a page's text well enough, but structured data goes further: explicitly labelling *what kind of thing* a piece of content is (a recipe, a product, an FAQ, an article) using a shared, standardised vocabulary — schema.org — that search engines specifically look for and reward with enhanced result displays.

What Structured Data Actually Buys You

Linux Performance Tuning — Free Course

osztromok.com › linux › performance

★★★★★ 4.8 rating · 8 chapters · Free

Learn to diagnose CPU, memory, disk, and network bottlenecks on Linux, scenario by scenario.

That star rating and metadata row is a "rich result" — search engines only show this for pages with valid structured data describing the content as a `Course` or similar type. Without it, the exact same page shows only as a plain text snippet, regardless of how good the actual content is.

JSON-LD — The Recommended Format

Schema.org data can be embedded three different ways historically (microdata attributes directly in HTML tags, RDFa, or JSON-LD) — Google explicitly recommends JSON-LD today, since it's cleanly separated from the visible markup and far easier to generate and validate correctly.

```
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "Course",
  "name": "Linux Performance Tuning",
```

```
  "description": "Learn to diagnose CPU, memory, disk, and network bottlenecks on Linux.",
  "provider": {
    "@type": "Organization",
    "name": "osztromok.com"
  }
}
</script>
```

This block lives in `<head>` (or anywhere in `<body>`) and is never displayed to visitors at all — it exists purely for search engine crawlers and other automated tools to read.

Microdata — The In-Markup Alternative

The older alternative embeds the same vocabulary directly as attributes on the visible HTML elements themselves — still valid, but generally considered messier to maintain than a separate JSON-LD block.

```
<div itemscope itemtype="https://schema.org/Course">
  <h2 itemprop="name">Linux Performance Tuning</h2>
  <p itemprop="description">Learn to diagnose CPU, memory, disk, and network bottlenecks.</p>
</div>
```

JSON-LD	Microdata
Separate block, doesn't touch visible markup, easiest to generate programmatically and validate independently. Generally the recommended default for new projects.	Attributes woven directly into existing tags — no duplication of content, but couples the structured data tightly to the visible HTML's exact structure.

Common schema.org Types Worth Knowing

- **Article / BlogPosting** — for written content, enables author/date display in results.
- **FAQPage** — for question/answer content, can produce an expandable FAQ display directly in search results.
- **Recipe** — enables rich recipe cards with cook time, ratings, and ingredient previews.

- **Product** — enables price, availability, and review star display.
- **BreadcrumbList** — shows a breadcrumb trail (Home › Linux › Performance) in the search result URL line itself, instead of a raw URL.
- **Course** — used in the example above, relevant directly to osztromok.com's actual content.

FAQPAGE PAIRS NATURALLY WITH CHAPTER 8'S DETAILS/SUMMARY ELEMENT

A visible FAQ built from `<details>` / `<summary>` pairs and a matching `FAQPage` JSON-LD block describing the same questions and answers is a genuinely common, complementary real-world pairing — the visible HTML serves users, the structured data serves search engines, both describing the identical content.

Validating Structured Data

Google's Rich Results Test (and the more general Schema.org Validator) checks a URL or pasted code against the actual schema definitions, flagging missing required properties and genuine syntax errors — directly analogous to Fundamentals Chapter 10's W3C validator, but specifically for structured data rather than HTML markup itself.

STRUCTURED DATA MUST ACCURATELY DESCRIBE CONTENT THAT'S GENUINELY VISIBLE ON THE PAGE

Search engines actively penalise structured data that doesn't match what a visitor would actually see — claiming a 5-star rating in JSON-LD with no actual reviews or rating system visible anywhere on the page is exactly the kind of mismatch that can trigger a manual action against a site. Structured data should describe real, visible content, never invented or exaggerated claims purely to win rich results.

CHAPTER 9 QUICK REFERENCE

- **Structured data** — explicitly labels content type using shared schema.org vocabulary, enabling rich search results
- **JSON-LD** — recommended format; a separate <script type="application/ld+json"> block, invisible to visitors
- **Microdata** — older alternative, woven directly into visible tags via itemscope/itemtype/itemprop
- **Common types:** Article, FAQPage, Recipe, Product, BreadcrumbList, Course
- **FAQPage + details/summary** — a natural, common real-world pairing (Chapter 8)
- **Validate with** Google's Rich Results Test or the Schema.org Validator

- **Never claim content that isn't genuinely visible on the page** — mismatched structured data can trigger search engine penalties
- **Next chapter:** forms + JavaScript — client-side validation hooks, FormData

CHAPTER 10: FORMS + JAVASCRIPT

COURSE 2 · CH 10

Forms + JavaScript

Hooking into the validation built-ins from Chapter 1, and reading submitted data with the FormData API

Chapter 1 of this course covered native browser validation (`required`, `pattern`, etc.) — entirely declarative, no JavaScript needed. This chapter covers where JavaScript genuinely adds value on top of that: custom validation messages, validating logic too complex for built-in attributes alone, and reading a form's data without manually grabbing every field one at a time.

The Constraint Validation API — JavaScript Meets Built-In Validation

Every form-related element exposes methods and properties that hook directly into the native validation from Chapter 1, rather than replacing it entirely.

PROPERTY/METHOD	WHAT IT DOES
`.checkValidity()`	Returns true/false — does this field currently satisfy its built-in constraints?
`.validity`	A detailed object describing exactly WHY it's invalid (valueMissing, patternMismatch, tooShort, etc.)
`.setCustomValidity(msg)`	Overrides the browser's default error message with your own custom text
`.reportValidity()`	Triggers the native validation UI to display immediately, as if the form had been submitted

```
// Custom validation message, still using the browser's native validation UI
const input = document.querySelector('#username');

input.addEventListener('invalid', () => {
  if (input.validity.valueMissing) {
    input.setCustomValidity('Please tell us what to call you!');
  } else {
    input.setCustomValidity(''); // clear it once valid again
  }
});
```

SETCUSTOMVALIDITY MUST BE CLEARED ONCE THE FIELD BECOMES VALID AGAIN

A custom message set once stays in effect indefinitely — even after the user fixes the field — unless explicitly cleared with `setCustomValidity('')`. Forgetting this step is the single most common bug with this API, leaving a field permanently marked invalid even when its actual value is now correct.

Validation Beyond What Built-In Attributes Can Express

Some rules genuinely can't be expressed with `pattern` or `min`/`max` alone — comparing two fields against each other is the classic example.

```
// Password confirmation — comparing TWO fields, impossible with built-in attributes alone
const form = document.querySelector('form');

form.addEventListener('submit', (event) => {
  const password = document.querySelector('#password');
  const confirm = document.querySelector('#confirmPassword');

  if (password.value !== confirm.value) {
    confirm.setCustomValidity('Passwords do not match');
    confirm.reportValidity();
    event.preventDefault(); // stop the actual submission
  }
});
```

FormData — Reading Submitted Values Without Grabbing Each Field

Rather than manually querying every input by ID and reading `.value` one at a time, `FormData` reads an entire form's submittable values at once, keyed by each field's `name` attribute (the exact attribute Fundamentals Chapter 7 established as essential for submission).

```
const form = document.querySelector('form');

form.addEventListener('submit', (event) => {
  event.preventDefault(); // handle it ourselves instead of a normal page navigation

  const data = new FormData(form);

  console.log(data.get('email'));    // the value of the field named "email"
  console.log(data.get('username')); // the value of the field named "username"
```

```
  // Iterate over every field at once:
  for (const [key, value] of data.entries()) {
    console.log(key, value);
  }
});
```

Sending FormData Without a Page Reload

```
const data = new FormData(form);

fetch('/api/submit', {
  method: 'POST',
  body: data
});
```

Passing a `FormData` object directly as a `fetch` request's body automatically sets the correct content type and encoding, including any file inputs (Fundamentals Chapter 5's `type="file"`) — no manual serialisation needed.

> **CLIENT-SIDE JAVASCRIPT VALIDATION IS STILL JUST A UX LAYER — COURSE 1'S SERVER-SIDE WARNING STILL APPLIES**
>
> Everything in this chapter runs entirely in the browser and can be bypassed exactly like the built-in HTML validation from Chapter 1 — disabled JavaScript, a direct API call, or a modified request all skip it completely. Every value still needs genuine server-side validation regardless of how much client-side JavaScript validation exists on top.

CHAPTER 10 QUICK REFERENCE

- **checkValidity()/validity/setCustomValidity()/reportValidity()** — hook into native validation rather than replacing it
- **Always clear setCustomValidity('')** once a field becomes valid again — the most common bug with this API
- **Use JS validation for logic built-in attributes can't express** — comparing two fields, cross-field rules
- **FormData** — reads all submittable values at once, keyed by each field's name attribute
- **new FormData(form)** passed directly as a fetch body — handles encoding and file uploads automatically

- **Client-side JS validation is still bypassable** — server-side validation of every value remains required regardless
- **Next chapter:** web components intro — custom elements & shadow DOM basics, the lead-in to Course 3

CHAPTER 11: WEB COMPONENTS INTRO

COURSE 2 · CH 11

Web Components Intro

Custom elements and shadow DOM basics — defining your own genuinely new HTML tags, with their own encapsulated styling

Every element covered across this entire series so far — div, section, dialog, details — was built into the browser already. Web Components let you define a genuinely **new** HTML element, with its own behaviour and (optionally) completely isolated CSS. This chapter is an intentionally gentle introduction; Course 3 (Advanced) returns to this topic in full depth.

Custom Elements — Defining Your Own Tag

```
class GreetingCard extends HTMLElement {
  connectedCallback() {
    this.innerHTML = `<p>Hello, ${this.getAttribute('name')}!</p>`;
  }
}

customElements.define('greeting-card', GreetingCard);
```

```
// Used in HTML exactly like any built-in tag, once defined
<greeting-card name="Philip"></greeting-card>
```

A CUSTOM ELEMENT'S TAG NAME MUST CONTAIN A HYPHEN

This is a hard requirement, not a style convention — `customElements.define('greeting', ...)` (no hyphen) fails outright. The hyphen is specifically how the browser distinguishes a custom element from any current or future standard HTML tag, exactly the same reasoning behind the reserved `data-` prefix from Chapter 7.

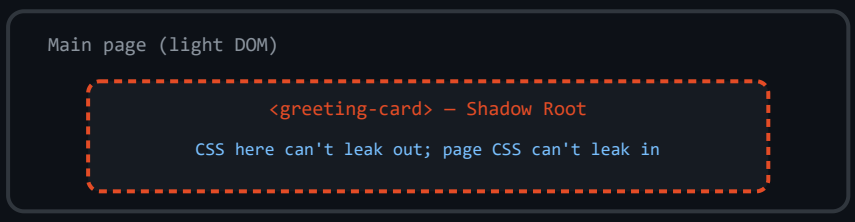
connectedCallback — When the Element Actually Runs Its Setup

`connectedCallback()` runs when the element is actually inserted into the document — not when the class is defined, and not necessarily the instant the HTML parses it. This timing

distinction matters once a custom element needs to read its own attributes or set up its initial content correctly.

Shadow DOM — Genuinely Isolated Internals

A custom element can attach a "shadow root" — a separate, encapsulated DOM subtree with its own scoped CSS that doesn't leak out to the rest of the page, and isn't affected by the rest of the page's CSS either.



The dashed boundary is the shadow root — a genuine, browser-enforced styling barrier

```
class GreetingCard extends HTMLElement {
  constructor() {
    super();
    const shadow = this.attachShadow({ mode: 'open' });
    shadow.innerHTML = `
      <style>
        p { color: red; font-weight: bold; } /* stays inside, never affects the rest of the p
      </style>
      <p>Hello!</p>
    `;
  }
}
```

Without Shadow DOM

A custom element's internal HTML is just regular, ordinary DOM — the page's global CSS can style it (intentionally or accidentally), and any internal class names could clash with classes used elsewhere on the page.

With Shadow DOM

A genuine, browser-enforced boundary — internal CSS never leaks out, page-level CSS never leaks in (with deliberate, limited exceptions). Eliminates an entire category of accidental style collision.

Why This Matters — The Underlying Motivation

- **True encapsulation** — a genuinely reusable widget (a custom date picker, a rating-stars component) that can't have its internal styling accidentally broken by whatever CSS happens to exist elsewhere on a page it's dropped into.
- **Framework-independent reuse** — a properly built web component works in a React app, a Vue app, or a plain HTML page identically, since it's a genuine browser feature rather than tied to any specific JavaScript framework.
- **Native, not a library dependency** — unlike component systems baked into specific frameworks, this is built directly into the browser itself, requiring no external library to function at all.

> **THIS IS GENUINELY AN INTRODUCTION, NOT THE FULL PICTURE**
>
> Real web components also use the `<template>` element from Chapter 8 (cloning a template's content into the shadow root is a common, cleaner pattern than building HTML via template literals as shown above), `<slot>` elements for letting users pass in their own content, and a fuller lifecycle than just `connectedCallback`. Course 3, Chapter 1 goes through all of this properly — this chapter exists purely to establish the core concept before that deeper dive.

CHAPTER 11 QUICK REFERENCE

- **customElements.define('tag-name', ClassName)** — registers a genuinely new HTML element
- **Custom element tag names MUST contain a hyphen** — a hard requirement, distinguishes them from standard/future HTML tags
- **connectedCallback()** — runs when the element is actually inserted into the document
- **attachShadow({mode: 'open'})** — creates a shadow root, a genuine CSS encapsulation boundary
- **Shadow DOM benefit:** internal CSS never leaks out, page CSS never leaks in — eliminates accidental style collisions
- **Framework-independent** — a real browser feature, works identically across React/Vue/plain HTML
- **Full depth (template/slot, full lifecycle) covered in Course 3, Chapter 1**
- **Next chapter (final, Course 2):** practical project — building a real contact form / project page

CHAPTER 12: PRACTICAL PROJECT - CONTACT FORM

COURSE 2 · CH 12 · FINAL CHAPTER · PROJECT

Practical Project — A Real Contact Form

Combining everything from this Intermediate course into one genuinely production-ready contact page

This final chapter builds a single, complete contact form page — deliberately combining validation, accessibility, structured data, and meta tags from across this entire course into one realistic artefact, rather than introducing anything new.

The Complete Page

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1"> // Ch5
  <title>Contact Us — osztromok.com</title>
  <meta name="description" content="Get in touch — questions, feedback, or just to say hello.
</head>
<body>

  <main>
    <h1>Contact Us</h1>

    <form action="/submit-contact" method="post" id="contactForm">

      <fieldset> // Ch1
        <legend>Your Details</legend>

        <label for="name">Name</label>
        <input type="text" id="name" name="name" required minlength="2"> // Ch1

        <label for="email">Email</label>
        <input type="email" id="email" name="email" required>
      </fieldset>

      <label for="message">Message</label>
```

```
      <textarea id="message" name="message" required minlength="10" rows="5"></textarea>

      <button type="submit">Send Message</button>
    </form>

    <details> // Ch8
      <summary>How long until I get a reply?</summary>
      <p>Usually within 2 business days.</p>
    </details>
  </main>

  <script type="application/ld+json"> // Ch9
  {
    "@context": "https://schema.org",
    "@type": "ContactPage",
    "name": "Contact Us"
  }
  </script>

  <script>
    // Ch10 — custom cross-field-style validation example: confirm before sending
    document.getElementById('contactForm').addEventListener('submit', (e) => {
      // FormData available here if a fetch-based submission were used instead
      // of the plain action/method POST shown above
    });
  </script>

</body>
</html>
```

What Each Decision Connects Back To

- **viewport + unique title/description** — Chapter 5, makes the page genuinely mobile-responsive and gives it a real search engine presence.
- **fieldset/legend around the personal details** — Chapter 1, groups related fields with a real accessibility benefit.
- **required + minlength on every field** — Chapter 1's built-in validation, with the understanding that server-side validation still happens on the receiving end regardless.
- **details/summary FAQ snippet** — Chapter 8, a JavaScript-free collapsible answer to a common question.

- **JSON-LD ContactPage markup** — Chapter 9, gives search engines explicit context about this page's purpose.
- **The submit listener stub** — Chapter 10, the hook point where FormData and custom validation logic would actually live for a fetch-based, no-reload submission.

Final Project Checklist

- [x] **Every input has a matching label, correctly associated**
  Fundamentals Ch7
- [x] **Required fields use built-in validation attributes, not JS alone**
  Chapter 1
- [x] **The form uses POST, never GET, since it submits personal data**
  Fundamentals Ch7
- [x] **Meta description and title are unique to this specific page**
  Chapter 5
- [x] **Server-side validation is assumed for every submitted value, regardless of client-side checks**
  Chapter 1 / Chapter 10

A "COMPLETE" FORM IS NEVER PURELY A CLIENT-SIDE ARTEFACT

Everything in this chapter's markup is the client-side half of a real contact form — the `action="/submit-contact"` endpoint still needs server-side code to actually receive, validate again, and act on the submission (sending an email, storing it in a database). This chapter intentionally stops at the HTML boundary, which is exactly the scope of this course.

CHAPTER 12 QUICK REFERENCE — AND COURSE 2 WRAP-UP

- **A real contact form combines:** viewport/meta tags (Ch5), fieldset/legend (Ch1), built-in validation (Ch1), details/summary (Ch8), structured data (Ch9), and a JS submit hook (Ch10)
- **Always POST, never GET,** for forms submitting personal data

- **Client-side markup is only half the picture** — a real server-side handler is still required
- **Course 2 recap:** forms in depth (Ch1) → tables in depth (Ch2) → multimedia (Ch3) → iframes (Ch4) → meta tags (Ch5) → accessibility (Ch6) → data attributes (Ch7) → HTML APIs (Ch8) → structured data (Ch9) → forms+JS (Ch10) → web components intro (Ch11) → real project (Ch12)
- **Next:** Course 3 — HTML Advanced (Web Components in depth, Canvas, SVG, drag-and-drop, File API, PWAs, and more)