HTML & CSS

A Clear Guide
to Modern Websites,
Responsive Design,
Accessibility,
Testing and Publishing

How Websites Work and Workspace Setup

A website can look simple on the screen, but several systems work together behind it. The browser requests resources, reads HTML, applies CSS, creates an internal document structure, calculates layout, and draws the result. More complex websites can also communicate with servers, databases, and external services.

This chapter explains that foundation. The aim is not to perform setup tasks. The aim is to understand the parts of a web project and the roles they play.

1.1 Websites, Webpages, Web Applications, and Static and Dynamic Content

A webpage is one document or view presented in a browser. A news article, product page, contact page, and account page can each be a webpage. A website is a collection of related webpages and resources under one identity. Those resources can include HTML documents, stylesheets, images, fonts, audio, video, and scripts.

A web application is a web-based product designed mainly for tasks and ongoing interaction. Webmail, online banking, booking systems, and project-management tools are common examples. The border between a website and a web application is not absolute. A single product can contain both information pages and application-like areas.

Static delivery means the server sends a resource that already exists. For example, a stored file named about.html can be sent in the same form to every visitor.

Dynamic generation means the server creates a response when a request arrives. An online store can read product information from a database, combine it with a template, and generate the HTML for that product page.

A real website can combine several approaches. A logo may be a static image, a product page may be generated on the server, and a shopping-basket total may change in the browser with JavaScript. A useful distinction is whether the final HTML was prepared before the request or generated because of the request.

1.2 Front End, Back End, Databases, APIs, and the Roles of HTML, CSS, and JavaScript

The front end is the part of a web product handled by the browser. It includes the interface that people see and operate.

HTML, CSS, and JavaScript have different responsibilities.

```
<h1>Community Workshop</h1>
<p>Learn basic bicycle maintenance.
</p>
```

HTML describes the structure and meaning of content. In this example, the first line is a main heading and the second line is a paragraph.

```
h1 {
    color: #205c32;
}
```

CSS controls presentation. It can change colour, typography, spacing, borders, layout, and responsive behaviour. It does not change the meaning of the h1 element.

JavaScript is a programming language that can add programmed behaviour. It can react to events, update content, perform calculations, and communicate with servers.

The back end is software that runs on a server. It can process requests, check permissions, work with databases, and return responses.

A database stores structured information. An online store may keep products, customers, and orders in a database.

An API, or Application Programming Interface, is a defined way for software systems to communicate. A weather website, for example, may request forecast data from a weather API and then present the result in its own interface.

HTML and CSS are essential front-end technologies, but they are not replacements for server logic, databases, or application programming.

1.3 Browsers, Rendering, the DOM, and How Source Files Become a Visible Page

A browser does more than display a file. It interprets several resources and turns them into a rendered page.

A simplified process is:

```
HTML source
    ↓
DOM
    +
CSS rules
    ↓
layout and painting
    ↓
visible page
```

The **DOM**, or Document Object Model, is a tree-like representation of the document created by the browser.

For example:

```
<main>
    <h1>Repair Guide</h1>
    <p>Bring one item.</p>
</main>
```

can be represented conceptually as:

```
main
├── h1
│    └── text
└── p
     └── text
```

The browser also reads CSS and determines which rules apply to which elements. It then calculates sizes and positions and paints pixels to the screen.

Browsers contain error-recovery rules. Incorrect HTML can still produce a visible page because the browser repairs or interprets the source as best it can. A visible result is therefore not proof that the source is correct.

1.4 Clients, Servers, Hosting, Domains, DNS, URLs, HTTP, and HTTPS

A **client** is software that requests a resource. A web browser is a common client.

A **server** is software or infrastructure that responds to requests. A server can return HTML, CSS, images, JSON data, downloads, and many other resources.

Hosting is the service or infrastructure that makes website resources available on a network.

A **domain name** is a human-readable name such as:

```
example.com
```

The **Domain Name System**, or **DNS**, connects domain names with the network information needed to reach services.

A **URL**, or Uniform Resource Locator, identifies a resource. For example:

```
https://example.com/guides/start.html
```

contains a scheme (`https`), a host (`example.com`), and a path (`/guides/start.html`).

HTTP is the protocol used for many web requests and responses. **HTTPS** is HTTP protected with TLS, which helps provide confidentiality, integrity, and server authentication during transport.

HTTP responses also contain status codes. Common examples include:

- `200 OK` - the request succeeded;
- `301 Moved Permanently` - the resource has a permanent redirect;
- `404 Not Found` - the requested resource was not found;
- `500 Internal Server Error` - the server encountered an error.

A status code describes the response. It does not describe the visual quality of the page.

1.5 Code Editors, Project Folders, Filenames, and Asset Organisation

A web project is easier to understand when its files have a clear structure.

A small project can look like this:

```
project/
├── index.html
├── style.css
├── images/
│   └── logo.svg
└── fonts/
    └── example.woff2
```

A **code editor** is software designed for working with source files. It can provide syntax colouring, search, indentation support, and other tools that make code easier to read.

The filename `index.html` has a special practical role on many web servers because it is commonly treated as the default document for a folder.

Consistent filenames reduce path errors. Web projects commonly use lowercase names and simple separators such as hyphens:

```
contact-us.html
team-photo.webp
main-navigation.css
```

Spaces, mixed letter case, and inconsistent names can make paths harder to manage, especially when a project moves between operating systems or servers with different case-sensitivity rules.

Folders also express relationships. Images belong in an image folder, fonts in a font folder, and related pages can be grouped in their own sections when the project becomes larger.

1.6 Local Servers, Developer Tools, and the Edit–Inspect–Validate–Test Workflow

A web browser can open a local HTML file directly, but a local HTTP server provides an environment that is closer to real web delivery. It gives resources HTTP URLs and makes it easier to observe request behaviour.

Browser **Developer Tools** provide information about the current page. They can expose the DOM, applied CSS, computed values, network requests, console messages, resource timing, and other browser data.

A professional development cycle can be described as:

```
edit source
    ↓
inspect browser result
    ↓
validate structure and syntax
    ↓
test behaviour
    ↓
correct problems
```

Each stage answers a different question.

Source editing changes the document. Browser inspection shows how the browser interpreted it. Validation checks rules of the language. Testing checks whether the page works in real conditions.

These forms of evidence should not be confused. A validator can confirm that markup follows syntax rules, but it cannot prove that a keyboard user can complete a task. A browser screenshot can show appearance, but it cannot prove that source structure is correct.

The main idea of this chapter is that a website is a system of related parts. HTML and CSS are central to the visible document, but browsers, servers, paths, protocols, files, and testing all affect the final result.

HTML Syntax and Document Structure

HTML gives a webpage structure and meaning. Its syntax is small enough to learn quickly, but correct structure matters because browsers, search tools, accessibility software, and other systems all depend on the document that HTML describes.

2.1 Elements, Tags, Content, Attributes, Values, Void Elements, and Nesting

The main structural unit in HTML is the **element**.

```
<h1>Community Skills Day</h1>
```

This element contains a start tag, text content, and an end tag.

```
<h1>      Community Skills Day
</h1>
start tag          content
end tag
```

An **attribute** adds information to an element.

```
<p class="intro">Entry is free.</p>
```

Here, `class` is the attribute name and `intro` is the value.

Elements can contain other elements. This is called **nesting**.

```
<section>
   <h2>Time and access</h2>
   <p>The event starts at
10:00.</p>
</section>
```

The `section` is the parent of the `h2` and `p`. The heading and paragraph are children of the section.

Correct nesting means inner elements close before their outer elements close.

Some HTML elements are **void elements**. They do not contain content and do not have end tags. Examples include `img`, `input`, `meta`, `link`, and `hr`.

The structure of an element comes from the HTML specification. A normal element cannot be turned into a void element simply by adding a slash.

2.2 Indentation, Whitespace, Character References, Comments, and Readable Source Code

HTML is processed by browsers, but it is also read by people. Clear formatting helps with review and maintenance.

Indentation shows nesting depth:

```
<main>
   <section>
      <p>Welcome.</p>
   </section>
</main>
```

Most ordinary whitespace in HTML text is collapsed by the browser. Several spaces or line breaks in the source usually appear as one space in normal rendered text.

HTML contains **character references** for characters that need a special written form in source code. For example:

```
&amp;
&lt;
&gt;
```

represent `&`, `<`, and `>`.

Comments use this syntax:

```
<!-- Editorial note -->
```

Comments can help explain source structure, but they are not private. They can be visible to anyone who can inspect the page source.

Readable source code uses consistent indentation, meaningful spacing, and comments only when they add useful information.

2.3 <!doctype html>, html, head, and body

A modern HTML document begins with a doctype:

```
<!doctype html>
```

This tells the browser to use standards-oriented HTML rendering rather than older compatibility modes.

The root element is `html`:

```
<html lang="en">
  ...
</html>
```

Inside it, the document is divided into `head` and `body`.

The **head** contains metadata and links to supporting resources. The **body** contains the content presented as the document itself.

The relationship can be represented conceptually as:

```
document
├── doctype
└── html
    ├── head  → metadata and linked resources
    └── body  → document content
```

The structure is important even when browsers can recover from missing or incorrect markup.

2.4 Character Encoding, Document Language, Text Direction, and Viewport Metadata

A document needs a known character encoding so that bytes are interpreted as the intended characters

UTF-8 is the standard choice for modern web pages:

```
<meta charset="utf-8">
```

The document language is declared on the `html` element:

```
<html lang="en">
```

The language information can help pronunciation, translation, spell checking, and other processing.

Text direction can be declared when needed:

```
<html lang="ar" dir="rtl">
```

`rtl` means right-to-left. `ltr` means left-to-right.

Responsive pages also commonly include viewport metadata:

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

This gives mobile browsers the expected starting relationship between the layout viewport and the device width. It does not make a rigid page responsive by itself.

2.5 Page Titles, Descriptions, Favicons, Stylesheet Links, and Essential Head Content

The `title` element provides the document title used by browsers and many other systems.

```
<title>Workshop Schedule | Riverside Centre</title>
```

A good title identifies the page clearly and can include the site name when useful.

A description can be provided with metadata:

```
<meta
    name="description"
    content="Workshop times and access information for Riverside Centre.">
```

Search services may use this description when presenting a result, but they are not required to display it exactly.

A favicon can be linked from the head:

```
<link rel="icon" href="favicon.svg" type="image/svg+xml">
```

An external stylesheet is linked with:

```
<link rel="stylesheet" href="style.css">
```

The `head` is therefore the place where the document identifies itself and connects to supporting resources that are not the main visible content.

2.6 HTML Validation, Nesting Errors, Obsolete Markup, and Correction Workflow

Browsers try to recover from incorrect HTML. A page can therefore appear normal even when the source contains errors.

A **validator** checks markup against the rules of HTML. It can find problems such as invalid nesting, duplicate identifiers, misplaced elements, and obsolete markup.

For example, this structure is incorrect:

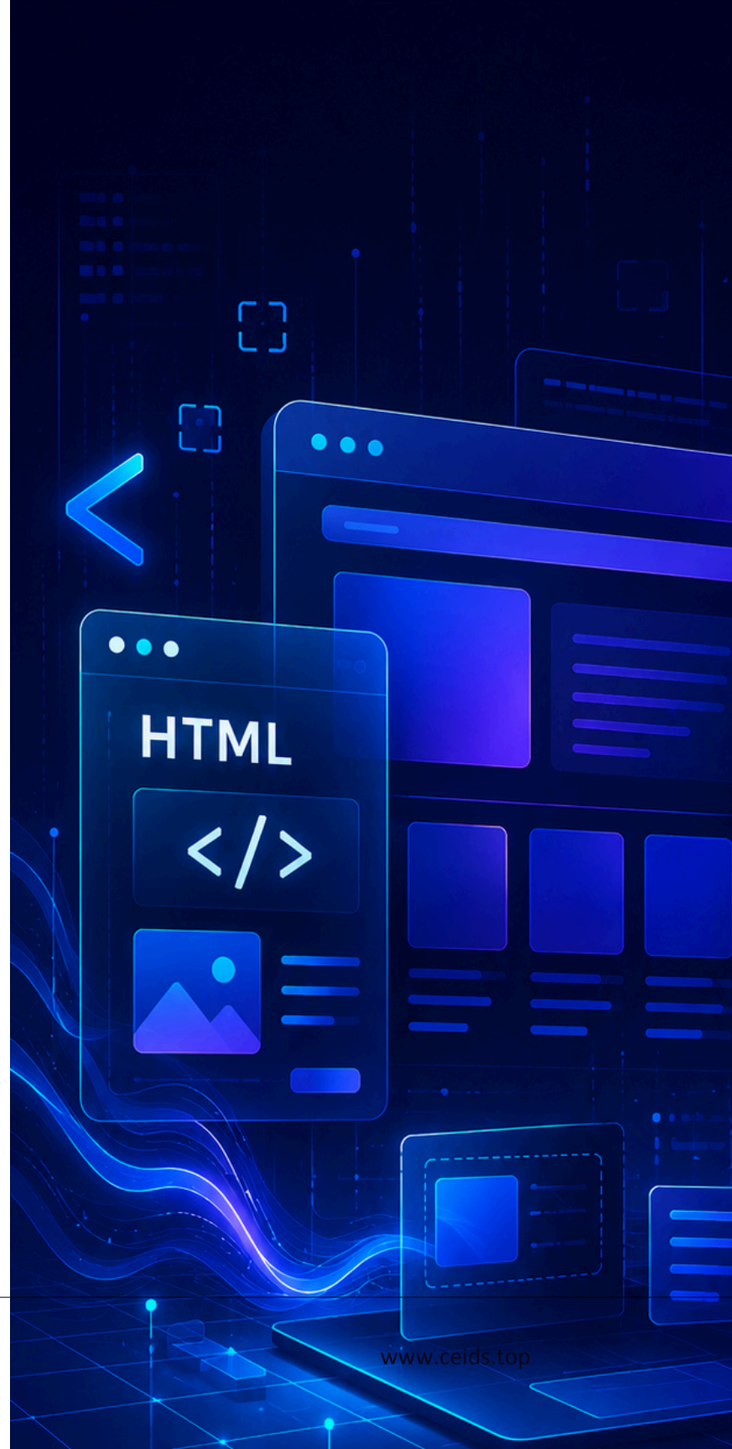
```
<p>
    Welcome
    <h2>Schedule</h2>
</p>
```

A heading does not belong inside a paragraph in this way.

Older HTML also contains elements and attributes that are now obsolete because presentation belongs in CSS. Examples from older pages include presentational markup such as `font` and layout-oriented use of HTML tables.

Validation is one part of quality checking. It can show that the source follows formal rules, but it does not prove that the wording is clear, the design is accessible, or the site behaves correctly in every browser.

The important principle is that HTML should describe the intended structure directly rather than depend on browser error recovery.



Text, Lists, and Semantic Content

HTML describes the meaning and relationships of written content. A heading, paragraph, quotation, date, list, and code sample can all look like text, but they represent different kinds of information.

3.1 Headings, Paragraphs, Thematic Breaks, and Logical Content Hierarchy

Headings represent levels in a document's structure.

```
h1   main subject
├── h2   major section
│    └── h3   subsection
└── h2   another major section
```

The number in `h1` to `h6` describes structural level, not visual size.

```
<h1>Community Workshop Guide</h1>
<h2>Before the workshop</h2>
<h3>What to bring</h3>
```

CSS can change the appearance later. HTML should preserve the logical hierarchy.

The `p` element represents a paragraph. A paragraph normally contains one connected idea or a small group of closely related sentences.

The `hr` element represents a **thematic break**. It can mark a meaningful change in topic or phase. Its default horizontal line is only a visual presentation of that meaning.

3.2 Meaningful Inline Text: strong, em, b, i, u, s, mark, small, sub, and sup

Inline elements add meaning or presentation inside a larger text passage.

`strong` represents strong importance:

```
<p><strong>Registration is required.
</strong></p>
```

`em` represents stress emphasis:

```
<p>Please arrive <em>before</em> the
session starts.</p>
```

`b` and `i` can represent text that needs attention or an alternate voice without the same meanings as `strong` and `em`.

`u` can represent text with a non-textual annotation, while `s` represents content that is no longer accurate or relevant.

`mark` represents highlighted relevance:

```
<p>Search result: <mark>accessible
forms</mark></p>
```

`small` represents side comments or small-print-like content such as copyright or legal notes.

`sub` and `sup` represent subscript and superscript, such as chemical formulas and mathematical notation:

```
H<sub>2</sub>O
x<sup>2</sup>
```

These elements should be chosen for meaning rather than for the browser's default visual style.

3.3 Ordered, Unordered, Description, and Nested Lists

An **unordered list** is appropriate when the sequence is not important.

```
<ul>
   <li>Notebook</li>
   <li>Pen</li>
   <li>Identification</li>
</ul>
```

An **ordered list** is appropriate when sequence or numbering matters.

```
<ol>
   <li>Choose a session.</li>
   <li>Check the time.</li>
   <li>Confirm the booking.</li>
</ol>
```

A **description list** represents name-value or term-description relationships.

```
<dl>
   <dt>DNS</dt>
   <dd>The system that connects
domain names with network
destinations.</dd>
</dl>
```

Lists can also be nested when one item contains a real sub-list. Nesting should express the information hierarchy, not create indentation only for appearance.

3.4 Quotations, Citations, Abbreviations, Dates, and Contact Information

Short quotations can appear inside a paragraph with `q`:

```
<p>The guide says <q>bring one item
only</q>.</p>
```

Longer standalone quotations can use `blockquote`.

A work title can be marked with `cite` when appropriate.

Abbreviations can use `abbr`:

```
<abbr title="Cascading Style
Sheets">CSS</abbr>
```

The `time` element can provide machine-readable date or time information:

```
<time datetime="2026-09-12">12
September 2026</time>
```

Contact information related to an author or organisation can use `address`. It does not simply mean any postal address appearing on a page.

The value of these elements is that software can understand the kind of information being presented, not only its visual appearance.

3.5 Code, Preformatted Text, Keyboard Input, Sample Output, and Variables

The `code` element represents computer code:

```
<code>display: grid;</code>
```

The `pre` element preserves whitespace and line breaks. It is useful for code or text where spacing is part of the presentation.

```
<pre><code>h1 {
   color: navy;
}</code></pre>
```

`kbd` represents user input from a keyboard or similar input system.

`samp` represents sample output from a program.
`var` represents a variable name in a technical or mathematical expression.

These elements help distinguish instructions, code, output, and variables from ordinary prose.

3.6 Semantic Page Regions, Containers, and Global Attributes

Semantic elements describe major regions and content structures.

`header` contains introductory or navigational material related to a page or section.

`nav` identifies a major navigation area.

`main` identifies the dominant content of the page.

`article` represents a self-contained composition such as a news article, post, or independent item.

`section` groups content around a related subject, usually with a heading.

`aside` represents content that is related but not central to the surrounding content.

`footer` contains closing information for a page or section.

`search` can identify a region containing search or filtering controls.

Generic containers also remain useful. `div` is a block-level generic container, while `span` is an inline generic container. They are appropriate when no more specific semantic element fits.

Global attributes can appear on many HTML elements. Important examples include `id`, `class`, `lang`, `dir`, `title`, `hidden`, and `data-*` attributes.

The central idea is that semantic HTML creates a meaningful document structure before CSS changes its appearance.

Links, Navigation, and Multipage Websites

Links connect web resources. A useful link has two important parts: a correct destination and clear text that helps a person understand where the link leads.

4.1 Anchor Anatomy, Destinations, Fragments, and Meaningful Link Text

Links are normally created with the `a` element.

```
<a href="about.html">About the workshop</a>
```

The `href` attribute contains the destination. The text between the tags is the visible link text.

A fragment link points to an element identified by `id`:

```
<a href="#schedule">Workshop schedule</a>
```

and:

```
<section id="schedule">
  <h2>Workshop schedule</h2>
</section>
```

The value after `#` must match the target `id`.

Link text should describe the destination. “Workshop schedule” is more informative than “Click here” because it makes sense regardless of whether the person uses a mouse, keyboard, touch screen, voice input, or assistive technology.

4.2 Absolute, Relative, Root-Relative, and Parent-Directory Paths

A **path** tells the browser where a resource is located.

An absolute URL includes the full address:

```
https://example.com/guides/start.html
```

A relative path is interpreted from the location of the current document:

```
about.html
images/photo.webp
```

A parent-directory path uses `..` to move one folder upward:

```
../index.html
```

A root-relative path starts with `/` and is resolved from the website's origin root:

```
/about/
```

These forms describe location relationships. The correct form depends on the project structure and deployment environment.

4.3 Linking Pages, Folders, Files, and Sections

A multipage site is a network of paths.

Consider this structure:

```
site/
├── index.html
├── about.html
└── guides/
    └── start.html
```

From `index.html`, the guide can be linked with:

```
<a href="guides/start.html">Start guide</a>
```

From `guides/start.html`, the home page is one level higher:

```
<a href="../index.html">Home</a>
```

Files such as PDFs, images, and text documents can also be link destinations.

Fragments can combine with paths:

```
<a href="about.html#team">Meet the team</a>
```

The browser loads the target document and then identifies the element with `id="team"`.

4.4 Email, Telephone, Download, and External Links; `target` and `rel`

A link can use special URL schemes.

Email:

```
<a href="mailto:info@example.com">Email the organiser</a>
```

Telephone:

```
<a href="tel:+441234567890">Call the organiser</a>
```

These links ask the user's environment to handle the action. They do not guarantee that an email or telephone application is available.

A link can point to a downloadable file:

```
<a href="guides/checklist.pdf">Preparation checklist</a>
```

The `download` attribute can suggest download behaviour for suitable same-origin resources, but browser and server behaviour still matter.

External links can use ordinary URLs. When a link intentionally opens a new browsing context with `target="_blank"`, `rel` values such as `noopener` can be relevant for security and behaviour. Opening new windows or tabs should be a deliberate choice rather than a default habit.

4.5 Primary Navigation, Secondary Navigation, Breadcrumbs, Skip Links, and Current-Page Indication

Navigation should express the organisation of a website.

Primary navigation usually contains the site's major destinations:

```
<nav aria-label="Primary">
  <a href="index.html">Home</a>
  <a href="about.html">About</a>
  <a href="contact.html">Contact</a>
</nav>
```

Secondary navigation supports a smaller area or topic.

Breadcrumbs show the path from a higher level to the current page:

```
Home > Guides > Accessibility
```

A **skip link** gives keyboard users a fast route past repeated navigation to the main content.

Current-page indication should communicate state clearly. Visual styling can help, but the information should not depend on colour alone. An attribute such as `aria-current="page"` can identify the current item when appropriate.

4.6 Multipage Structure, Broken Links, and Path Errors

A multipage website depends on stable relationships between files.

Common causes of broken links include:

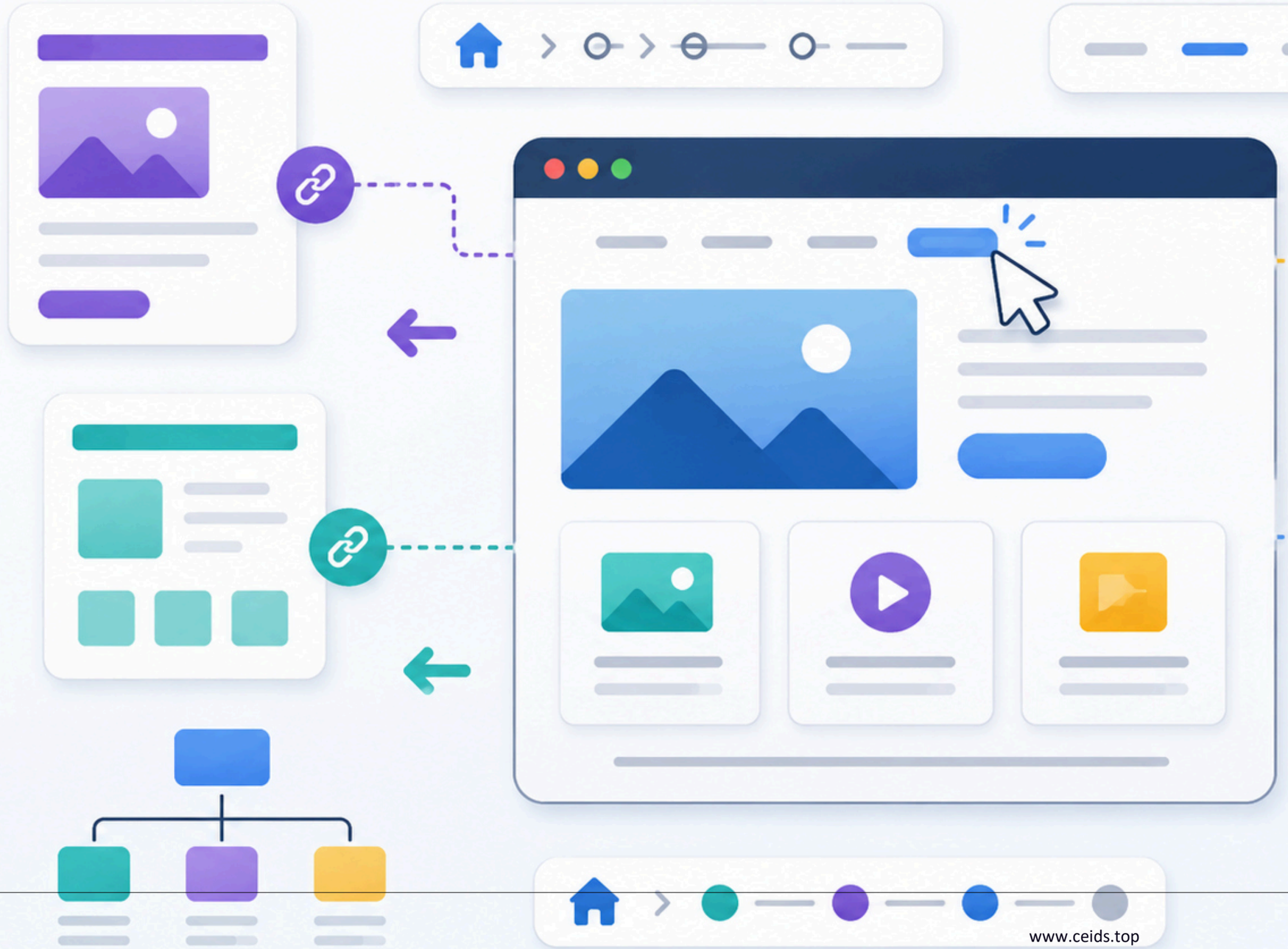
- wrong filenames;
- incorrect folder depth;
- incorrect letter case;
- moved files;
- changed IDs;
- incorrect root-relative assumptions;
- deployment under a different base path.

A link can be syntactically valid while still leading to the wrong content. Correct path syntax and

correct destination meaning are separate concerns.

A well-structured multipage site keeps navigation consistent, uses predictable filenames, and preserves stable identifiers where links depend on them.

The main principle is that navigation is both technical and human. Paths connect resources, while labels and structure help people understand those connections.



Images, Graphics, Audio, Video, and Embeds

Web pages can contain photographs, diagrams, icons, audio, video, captions, transcripts, and embedded documents. Good media use depends on more than visual appearance. File format, dimensions, alternative text, loading behaviour, licensing, accessibility, and privacy all matter.

5.1 Raster and Vector Graphics; JPEG, PNG, WebP, AVIF, GIF, and SVG

A **raster image** stores a grid of pixels. Photographs are typical raster images. If a raster image is enlarged far beyond its original detail, it can look soft or pixelated.

A **vector image** describes shapes mathematically. SVG is the main vector format used on the web. It is suitable for logos, icons, diagrams, and simple illustrations because it can scale without the same pixel enlargement problem.

Different formats have different strengths.

JPEG is widely used for photographs. It normally uses lossy compression and does not support ordinary alpha transparency.

PNG uses lossless compression and supports transparency. It is often suitable for screenshots, interface graphics, and images with transparent backgrounds.

WebP can support lossy and lossless compression, transparency, and animation.

AVIF can provide efficient compression and supports modern image features, but the best result depends on the source image and encoder settings.

GIF is historically associated with animation, but it has a limited colour model and is usually not the best format for modern full-colour photographs.

SVG stores vector graphics as markup. Because SVG can contain more than simple drawing information, untrusted SVG content should be treated carefully.

Changing a filename extension does not convert a file. A real export or conversion process must create data in the new format.

5.2 The img Element, Source, Alternative Text, Dimensions, and Aspect Ratio

The `img` element embeds an image:

```
<img
    src="images/workshop.jpg"
    alt="People repairing bicycles at
a community workshop"
    width="1200"
    height="800">
```

`src` identifies the image resource.

`alt` provides a text alternative when the image contributes information.

`width` and `height` can describe the image's intrinsic proportions. This information helps the browser reserve space before the image finishes loading and can reduce unexpected layout movement.

An image with dimensions `1200 × 800` has an aspect ratio of `3:2`.

Responsive CSS can allow an image to shrink while keeping its proportions:

```
img {
    max-width: 100%;
    height: auto;
}
```

The HTML dimensions and responsive CSS solve different problems. One supplies useful intrinsic information. The other controls rendered size in the layout.

5.3 Alternative Text for Informative, Decorative, Functional, and Complex Images

Alternative text depends on the purpose of an image in its context.

An **informative image** communicates content. Its alternative text should express the useful information.

```
<img
    src="entrance.jpg"
    alt="Step-free entrance beside
the main library doors">
```

A **decorative image** adds no information and can use an empty alternative text value:

```
<img src="pattern.svg" alt="">
```

A **functional image** is part of an interactive control. The alternative text should describe the function, not merely the picture.

A complex diagram may need a short alternative text plus a nearby detailed explanation in ordinary text.

The same image can need different alternative text in different contexts. Alternative text describes purpose, not the file itself.

5.4 figure, figcaption, Image Links, Licensing, and Attribution

The `figure` element can group self-contained content with a caption.

```
<figure>
    <img src="map.svg" alt="Walking
route from the station to the
centre">
    <figcaption>Recommended walking
route from Central Station.</
figcaption>
</figure>
```

`figcaption` provides a visible caption. It is not a replacement for useful alternative text when the image itself needs a text alternative.

An image can also appear inside a link. In that situation, the accessible name of the link needs to communicate the destination or action.

Media also has legal and ethical requirements. A file being publicly visible on the internet does not mean it is free to reuse.

Licensing can define whether a work may be copied, modified, redistributed, or used commercially. Some licences require attribution. Good attribution identifies the work, creator, source, and licence when those details are required.

5.5 Responsive Images with srcset, sizes, and picture; Basic SVG Use

A large image file may be unnecessary on a narrow screen. Responsive-image features allow the browser to choose among suitable candidates.

```
<img
   src="photo-800.jpg"
   srcset="photo-480.jpg 480w,
           photo-800.jpg 800w,
           photo-1200.jpg 1200w"
   sizes="(min-width: 60rem) 50vw,
92vw"
   alt="Students working at a
design workshop">
```

`srcset` describes available image candidates and their intrinsic widths.

`sizes` describes the expected rendered source size under different layout conditions.

The browser makes the final selection. Candidate choice can depend on viewport size, pixel density, zoom, cache state, and browser policy.

The `picture` element can provide different image formats or art-directed sources:

```
<picture>
   <source srcset="photo.avif"
type="image/avif">
   <source srcset="photo.webp"
type="image/webp">
   <img src="photo.jpg"
alt="Workshop participants">
</picture>
```

SVG can be embedded as an image file or included inline when direct styling or document interaction is genuinely needed. Inline SVG should come from a trusted source.

5.6 Accessible Audio, Video, Captions, Transcripts, Tracks, Iframes, and Privacy

Native audio and video elements can provide browser controls.

```
<video controls width="960"
height="540">
   <source src="arrival.mp4"
type="video/mp4">
   <track
      kind="captions"
      src="arrival.en.vtt"
      srclang="en"
      label="English">
</video>
```

Captions represent spoken content and important sounds for people who cannot hear the audio clearly or at all.

A transcript provides the spoken and relevant non-spoken information as text. Transcripts can also help search, translation, scanning, and users who prefer reading.

The `track` element can connect timed text such as captions to media.

An iframe embeds another document:

```
<iframe
   src="map.html"
   title="Walking route to the
learning centre">
</iframe>
```

The `title` helps identify the iframe's purpose.

Embedded third-party content can introduce privacy concerns because loading it may contact another service and reveal technical information about the visit. Third-party embeds can also fail, change, or become unavailable. Important information should therefore not depend entirely on an external frame when a normal text or link alternative is possible.

The main principle is that media should support the document rather than create unnecessary barriers, downloads, legal problems, or privacy risks.

Tables and Forms

Tables and forms solve different problems. Tables describe relationships in structured data. Forms allow people to enter or choose information for submission.

Both require correct semantics because visual appearance alone does not express their relationships.

6.1 Recognising Tabular Data and Why Tables Are Not Page-Layout Tools

A table is appropriate when data has meaningful relationships across rows and columns.

Workshop	Time	Room
Photography	09:00	Studio A
Cooking	10:30	Kitchen

Each row is one record. Each column represents the same field across records.

A person can read across a row to understand one workshop or down a column to compare times or rooms.

Ordinary page content placed side by side does not automatically become tabular data. Two articles, two navigation panels, or two columns of marketing text should normally be laid out with CSS rather than placed inside a table.

A useful question is whether the row and column relationships carry real meaning. If they do not, a table is probably the wrong structure.

6.2 Rows, Cells, Headers, Captions, Table Sections, and scope

The `table` element contains tabular data.

`tr` represents a row.

`th` represents a header cell.

`td` represents a data cell.

A simple table can look like this:

```
<table>
   <caption>Workshop schedule</
caption>
   <tr>
      <th scope="col">Workshop</th>
      <th scope="col">Time</th>
   </tr>
   <tr>
      <th scope="row">Photography</
th>
      <td>09:00</td>
   </tr>
</table>
```

The `caption` gives the table a visible title or explanation.

`scope="col"` identifies a column header. `scope="row"` identifies a row header in straightforward tables.

Large tables can also use `thead`, `tbody`, and `tfoot` to group rows by role.

The important point is that bold text does not create a header relationship. The `th` element and its structural position do.

6.3 Complex and Large Tables: Spans, Readability, Narrow Screens, Zoom, and Keyboard Access

Some data tables need cells that span several columns or rows.

`colspan` allows a cell to cover more than one column.

`rowspan` allows a cell to cover more than one row.

These features can represent real grouped data, but complicated spanning patterns also make tables harder to understand and maintain.

On narrow screens, a genuine data table may need local horizontal scrolling because its two-dimensional relationships cannot always collapse into one narrow column without losing meaning.

A common presentation pattern is a scrollable wrapper around the table:

```
.table-wrap {
   overflow-x: auto;
}
```

This keeps horizontal scrolling local to the table instead of forcing the entire page to scroll sideways.

Zoom and text enlargement can make a table wider than expected. Good table design therefore depends on readable headers, sensible content length, and testing under constrained space.

6.4 Form Purpose, Submission Basics, action, GET and POST, name, value, Labels, and Buttons

A form collects information and sends it for processing.

```
<form action="/search" method="get">
   <label for="q">Search</label>
   <input id="q" name="q"
type="search">
   <button type="submit">Search</
button>
</form>
```

`action` identifies the destination that receives the submission.

`method` describes how form data is sent.

With **GET**, submitted values normally become part of the URL query. This is suitable for actions such as search and filtering when the request is safe to repeat and bookmark.

With **POST**, form data is sent in the request body. POST is commonly used for operations that change data or should not place the submitted values directly in the URL.

The `name` attribute identifies a field in submitted data.
The `value` is the submitted value associated with that name.

A visible `label` identifies the purpose of a form control.
A button with `type="submit"` submits the form using the form's configured behaviour.

HTML form markup can describe the request, but secure processing and final validation require server-side logic.

6.5 Text Fields, Specialised Inputs, Textareas, Selects, Datalists, Checkboxes, Radios, and File Inputs

HTML provides different controls for different kinds of information.

`type="text"` is a general single-line text field.
`type="email"`, `type="url"`, `type="tel"`, `type="number"`, `type="date"`, and other specialised input types can provide additional semantics and browser behaviour.

A `textarea` is suitable for multi-line text.

A `select` presents a set of options.

A `datalist` can provide suggestions for some input controls while still allowing typed values.

Checkboxes represent independent on/off choices. Radio buttons represent one choice from a group and share the same `name` value.

A file input allows the user to select a local file for submission.

The control type should match the nature of the information. Visual style should not be the main reason for choosing a control.

6.6 Fieldsets, Legends, Autocomplete, inputmode, Native Validation, Errors, Privacy, and Server Validation

Related form controls can be grouped with `fieldset` and described with `legend`.

```
<fieldset>
   <legend>Preferred  contact
method</legend>
   ...
</fieldset>
```

`autocomplete` can describe the purpose of common personal-information fields so browsers and assistive tools can recognise them.

`inputmode` can suggest a suitable on-screen keyboard layout without changing the underlying data type.

HTML also provides native constraints such as `required`, `minlength`, `maxlength`, `min`, `max`, and `pattern` in appropriate situations.

Native browser validation is useful, but it is not a security boundary. A server must still validate submitted data because browser-side controls can be bypassed or altered.

Error messages should explain the problem in words. Colour can support the message, but it should not be the only signal.

Form design also has a privacy dimension. A form should request only information that the service genuinely needs. Sensitive information needs appropriate handling beyond HTML and CSS.

The main principle is that forms should communicate clear relationships between questions, controls, values, and processing expectations.

CSS Syntax, Selectors, and the Cascade

CSS controls the presentation of structured documents. It can change colour, spacing, typography, borders, dimensions, layout, and responsive behaviour while HTML continues to provide meaning and structure.

7.1 Rules, Selectors, Declaration Blocks, Properties, Values, and Comments

A basic CSS rule looks like this:

```
body {
    background-color: #f2f6f4;
}
```

`body` is the **selector**. It identifies which elements the rule can match.

The braces contain the **declaration block**.

`background-color` is the **property**.

`#f2f6f4` is the **value**.

A stylesheet can contain many rules, each describing a presentation decision.

CSS comments use:

```
/* Shared colours for service
notices */
```

Comments can explain useful context, but stylesheets are normally visible to users, so comments should never contain secrets.

7.2 External, Internal, and Inline CSS

CSS can be connected to HTML in several ways.

An **external stylesheet** is a separate file linked from the document:

```
<link rel="stylesheet" href="style.
css">
```

This is the usual approach for reusable site styles because many pages can share one stylesheet.

Internal CSS appears in a `style` element inside the document head:

```
<style>
    p { color: #333; }
</style>
```

It can be useful for a small standalone document or a tightly contained example.

Inline CSS appears in an element's `style` attribute:

```
<p style="color: #333;">Example
text</p>
```

Inline styles are strongly tied to individual elements and are harder to reuse and maintain. They also participate differently in the cascade.

The method should be chosen according to scope and maintainability rather than convenience alone.

7.3 Type, Class, ID, Universal, Grouped, Combinator, and Attribute Selectors

A **type selector** matches elements by name:

```
p {
    line-height: 1.6;
}
```

A **class selector** matches elements by class:

```
.notice {
    border: 1px solid #555;
}
```

An **ID selector** uses `#`:

```
#main-content {
    max-width: 70rem;
}
```

IDs are unique document identifiers. Classes are usually more suitable for reusable styling.

The universal selector `*` matches all elements within its scope.

Grouped selectors share declarations:

```
h1,
h2,
h3 {
    line-height: 1.2;
}
```

Combinators describe relationships. For example:

```
nav a {
    text-decoration-thickness: 0.1em;
}
```

matches links that are descendants of `nav`.

Attribute selectors can match based on attributes:

```
input[type="email"] {
    border-color: #315b69;
}
```

Selectors should be as simple as the required relationship allows.

7.4 Pseudo-Classes, Pseudo-Elements, Inheritance, and Browser Default Styles

A **pseudo-class** represents a state or structural condition.

```
a:hover {
    text-decoration-thickness: 0.15em;
}
a:focus-visible {
    outline: 3px solid #111;
}
```

A **pseudo-element** represents a generated or conceptual part of an element.

```
li::marker {
    font-weight: 700;
}
```

Some CSS properties are inherited. For example, text colour often passes from a parent to descendants unless another rule changes it.

Other properties, such as borders and margins, usually do not inherit.

Browsers also provide **user-agent stylesheets**. These default rules make headings, links, lists, buttons, and form controls usable before author CSS is added.

Author CSS works with, overrides, or extends those defaults.

7.5 Cascade Order, Specificity, Source Order, IDs, and `!important`

The word **cascading** describes how CSS resolves competing declarations.

When more than one declaration could set the same property, CSS considers several factors. At a beginner level, a useful simplified sequence is:

```
applicability
     ↓
origin and importance
     ↓
specificity
     ↓
source order
```

Specificity is a weighting system based on selector structure. An ID selector has more specificity than a class selector, and a class selector has more specificity than a type selector.

For example:

```
p { color: black; }
.notice { color: darkred; }
```

A paragraph with class `notice` matches both rules, but the class selector is more specific.

If two competing rules have equal specificity and the same origin and importance, the later rule usually wins by source order.

Heavy use of IDs for styling can make overrides difficult. Frequent use of `!important` can make the cascade harder to understand and maintain. Both are tools, but they should not replace clear stylesheet structure.

7.6 Computed Styles, Conflicts, and Readable Selectors

The browser calculates a final **computed value** for each property on each element.

Developer tools can show:

- which selectors matched;
- which declarations were overridden;
- which values were inherited;
- the final computed result.

A visual problem is often easier to understand when the competing declarations are inspected rather than guessed.

For example, a margin may appear to be ignored because another rule resets it, or a colour may come from inheritance rather than the element's own rule.

Readable selectors reduce this complexity. A selector such as:

```
.card-title {
   margin-block-start: 0;
}
```

is usually easier to maintain than a deeply chained selector that depends on many ancestors.

The main principle is that the cascade is predictable. CSS becomes easier to debug when selector scope, specificity, inheritance, and source order are understood together.

Colours, Units, Sizing, and the Box Model

CSS values describe the requested presentation of an element. The meaning of a value depends on the property that receives it, the surrounding layout, and sometimes the size of the viewport or another element.

8.1 CSS Value Types, Keywords, Numbers, Strings, URLs, Percentages, and Invalid Declarations

Each CSS property accepts a defined set of value types.

Examples include keywords, numbers, lengths, percentages, colours, strings, URLs, and functions.

```
line-height: 1.5;
```

uses a unitless number.

```
padding: 1rem;
```

uses a length.

```
width: 90%;
```

uses a percentage.

A value that is valid for one property may be invalid for another.

CSS also has general keywords such as `initial`, `inherit`, and `unset`. Their meaning depends on the property and inheritance rules.

If a declaration has invalid syntax or an unacceptable value, the browser normally ignores that declaration rather than making the whole stylesheet unusable.

8.2 px, %, em, rem, ch, and Viewport Units

A CSS **pixel**, written `px`, is a reference unit used by CSS. It is not always equal to one physical screen pixel.

Percentages are relative values, but the reference depends on the property. A percentage width usually relates to an appropriate containing block. Other percentage properties can use different references.

`em` is relative to font sizing in the relevant context. `rem` is relative to the computed font size of the root element.

`ch` is based on the advance measure of the `0` glyph in the current font and can be useful for approximate text measure.

Viewport units relate to viewport dimensions. Examples include `vw`, `vh`, `svh`, `lvh`, and `dvh`.

Different units are useful for different relationships. There is no single “best” unit for all CSS.

8.3 Hex, RGB, HSL, Alpha, currentColor, and CSS Math Functions

CSS supports several standard colour notations.

Hexadecimal:

```
color: #1f4f5f;
```

RGB:

```
color: rgb(31 79 95);
```

HSL:

```
color: hsl(192 51% 25%);
```

Alpha transparency can also be included in modern colour syntax.

`currentColor` represents the current value of the `color` property and can help borders or icons follow text colour.

CSS mathematical functions can express relationships.

```
width: calc(100% - 2rem);
```

`min()`, `max()`, and `clamp()` can choose values according to limits.

```
font-size: clamp(2rem, 4vw, 3rem);
```

The result stays between the stated minimum and maximum while the preferred value can change.

8.4 Content, Padding, Borders, Margins, Margin Collapse, gap, and box-sizing

Every visible element can be understood through the **box model**.

```
margin
└── border
    └── padding
        └── content
```

The **content box** contains text, images, or child elements.

Padding creates space between content and border.

Border surrounds the padding and content.

Margin creates space outside the border.

Margins can collapse in some normal block-flow situations. This is a specific CSS behaviour and does not apply in the same way to Flexbox and Grid layout.

`gap` creates spacing between items in layout systems such as Flexbox and Grid.

By default, declared width and height normally refer to the content box. With:

```
* {
    box-sizing: border-box;
}
```

declared width and height include padding and borders in the box-sizing calculation.

8.5 Width, Height, Minimum and Maximum Sizes, Intrinsic Sizing, aspect-ratio, object-fit, and object-position

CSS sizing can be fixed, flexible, or content-driven. `width` and `height` request sizes.

`min-width`, `max-width`, `min-height`, and `max-height` apply constraints.

Ordinary text content should usually be allowed to grow vertically. Fixed heights can cause clipping or overflow when content becomes longer, is translated, or is enlarged.

Intrinsic sizing concepts describe sizes based on content. Terms such as `min-content`, `max-content`, and `fit-content` express different relationships between content and available space.

`aspect-ratio` describes a preferred proportional relationship:

```
.media-frame {
    aspect-ratio: 16 / 9;
}
```

For replaced media such as images, `object-fit` controls how content fits inside a set box.

`cover` fills the box and may crop.

`contain` keeps the complete object visible and may leave unused space.

`object-position` controls the alignment of the object within that box.

8.6 Overflow, Scrolling, Clipping, Custom Properties, and Box-Model Debugging

Overflow occurs when content needs more space than its box provides.

Possible values include `visible`, `hidden`, `clip`, `auto`, and `scroll` in appropriate contexts.

A scroll container can be useful for genuinely wide content such as a data table. Hiding overflow simply to remove an unwanted scrollbar can conceal content and focus indicators.

CSS custom properties store reusable values:

```
:root {
    --space-medium: 1rem;
}
```

They can be used with `var()`:

```
.card {
    padding: var(--space-medium);
}
```

Custom properties participate in the cascade and usually inherit.

When a box behaves unexpectedly, the useful questions include:

- What is its containing block?
- Which sizing rule applies?
- Does padding or border affect the total size?
- Is a minimum size preventing shrinkage?
- Is content creating overflow?
- Is a layout system adding `gap` or alignment?

The main principle is that box sizing should describe useful relationships rather than force content into arbitrary dimensions.

Typography and Visual Styling

Typography and visual styling shape how information is presented. They support hierarchy, readability, grouping, and interaction, but they do not replace HTML meaning.

A heading is not a heading because it is large. A warning is not a warning because it is yellow. A disabled control is not disabled because it looks grey. HTML provides meaning and behaviour; CSS supports that meaning visually.

9.1 Typeface Categories, System Font Stacks, Fallbacks, and Responsible Font Selection

A **typeface** is a visual design for written characters. A **font family** groups related faces, such as regular, bold, and italic. A **font face** is one specific combination of family, weight, and style. A **font file** contains the data used to draw glyphs.

CSS can request an ordered list of fonts:

```
body {
   font-family: system-ui, sans-
serif;
}
```

This is a **font stack**. The browser tries the available choices in order.

Generic families include:

- `serif`;
- `sans-serif`;
- `monospace`;
- `system-ui`.

These names do not identify one exact font. The actual typeface can differ by operating system and browser.

Font choice also depends on language coverage. A typeface that contains English letters may not contain every Polish, Greek, Arabic, or other required character. The browser can fall back to another font for missing glyphs.

Responsible font selection also includes licensing. A font file being available on a computer or website does not automatically give permission for web distribution.

9.2 Font Family, Size, Weight, Style, Line Height, Line Length, Spacing, Alignment, and Rhythm

Readable typography depends on several connected properties.

`font-family` selects a family or stack.

`font-size` controls text size.

`font-weight` controls weight, such as normal or bold.

`font-style` can request values such as normal, italic, or oblique.

`line-height` controls vertical spacing between lines.

A unitless value is often useful because it scales with the element's font size:

```
body {
   line-height: 1.6;
}
```

Long lines can be difficult to scan. A text measure can be limited with `ch`:

```
.article-text {
    max-width: 65ch;
}
```

The `ch` unit gives an approximate measure based on the current font. It is not an exact character count.

`letter-spacing` changes spacing between characters. `word-spacing` changes spacing between words. Both should be used carefully because excessive values can reduce readability.

`text-align: start` follows the writing direction and is often more flexible than permanently choosing left or right alignment.

Vertical rhythm comes from consistent relationships among line height, headings, paragraphs, and section spacing. Repeated `<br>` elements should not be treated as a general spacing system.

9.3 Web Fonts, @font-face, WOFF2, Fallbacks, and font-display

A web font is a font resource downloaded by the browser as part of a website.

`@font-face` describes a downloadable face:

```
@font-face {
    font-family: "Example Sans";
    src: url("example-regular.woff2")
format("woff2");
    font-weight: 400;
    font-style: normal;
    font-display: swap;
}
```

WOFF2 is a modern web-font format designed for web delivery.

Separate font files can represent different weights or styles. The metadata should match the actual face. A regular font file should not be declared as bold simply to make the browser treat it as bold.

Fallback fonts remain important:

```
body {
    font-family: "Example Sans",
system-ui, sans-serif;
}
```

If the custom resource fails, the browser can continue with another suitable family.

`font-display` influences how fallback text and downloaded fonts are handled during loading. Values include `auto`, `block`, `swap`, `fallback`, and `optional`.

A font swap can change line wrapping because two typefaces can have different character widths. Web-font loading can therefore affect layout as well as appearance.

9.4 Backgrounds, Borders, Outlines, Radii, Separators, and Focus Rings

Background styling can include colour and images:

```
.notice {
    background-color: #fff4cc;
    background-image: url("pattern.
svg");
}
```

The background colour remains useful if the decorative image cannot load.

Essential information should not exist only inside a CSS background image because that image can fail, be hidden, or become difficult to interpret.

Borders form part of the box model:

```
.card {
    border: 2px solid #5f6f75;
    border-radius: 0.5rem;
}
```

`border-radius` changes the visual shape of corners.

It does not change the element's semantic role.

An **outline** is different from a border. Outlines do not normally occupy layout space and are commonly used for focus indication.

```
:focus-visible {
   outline: 3px solid #111;
   outline-offset: 3px;
}
```

Visible focus is essential for keyboard users. Removing the browser's focus indicator without a suitable replacement can make an interface difficult or impossible to operate.

9.5 Gradients, Shadows, Opacity, and Restrained Visual Effects

A CSS gradient is an image value generated by CSS.

```
.banner {
   background-color: #dbeafe;
   background-image:
      linear-gradient(135deg,
#dbeafe, #e0f2fe);
}
```

The solid colour provides a fallback and can also support the gradient visually.

Radial gradients spread outward from a centre or focal point.

`box-shadow` can create visual depth:

```
.card {
   box-shadow: 0 0.3rem 1rem rgb(0
0 0 / 0.12);
}
```

A shadow does not reserve layout space and should not be the only way to communicate an important boundary.

`text-shadow` can be decorative, but it should not be treated as a substitute for adequate text contrast.

`opacity` affects the entire rendered element, including its descendants:

```
.panel {
   opacity: 0.6;
}
```

An alpha colour changes only that colour value, while `opacity` changes the whole element as a group.

Visual effects are most useful when they strengthen hierarchy and state without creating distraction or reducing legibility.

9.6 Styling Lists, Tables, Buttons, and Form Controls Without Removing Usability

CSS can change the appearance of native elements while their HTML meaning remains intact.

List markers can be styled:

```
.features li::marker {
   font-weight: 700;
}
```

Removing list markers from every list can make list structure less obvious, especially when spacing is also small.

Tables can use borders, spacing, and background variation while retaining their real `table`, `th`, and `td` structure.

Buttons should remain real buttons:

```
<button type="submit">Send
registration</button>
```

CSS can change the visual style, but it should preserve clear states such as hover, focus, active, and disabled where those states exist.

Form controls often benefit from inheriting the surrounding typography:

```
button,
input,
select,
textarea {
   font: inherit;
}
```

A select element can be styled, but removing its native appearance and rebuilding all expected behaviour is a more advanced task.

Error styling should not depend on red colour alone. An invalid field can combine visible text, appropriate HTML or ARIA state, and a border or other visual cue.

The main principle is that visual styling should support meaning and usability rather than replace them.

Flexbox and CSS Grid

CSS layout begins with **normal flow**. Flexbox and Grid add more control when boxes need specific relationships. Positioning solves a different class of problems and should not replace ordinary page layout.

10.1 Normal Flow, Block and Inline Behaviour, Display Roles, Wrappers, and Content-Driven Layout

In normal flow, block content usually follows other block content vertically, while inline content participates within lines.

A heading followed by a paragraph already forms a layout:

```
<h1>Community Repair Café</h1>
<p>Bring one household item for a repair check.</p>
```

The paragraph follows the heading, and both grow when their content grows.

This content-driven behaviour is important. If a heading wraps onto two lines, later content moves down naturally.

CSS `display` values can change layout roles. Common introductory values include `block`, `inline`, and `inline-block`.

A **wrapper** is a generic container that applies a shared layout constraint:

```
.wrapper {
    width: 92%;
    max-width: 68rem;
    margin-inline: auto;
}
```

This allows a page region to shrink on narrow screens while limiting excessive growth on wide screens.

Fixed heights are risky for ordinary text content because translations, zoom, and longer text can require more vertical space.

10.2 Flex Containers, Items, Axes, Direction, and Wrapping

Flexbox is designed mainly for one-dimensional relationships.

```
.navigation-list {
    display: flex;
}
```

The element becomes a **flex container**. Its in-flow direct children become **flex items**.

Flexbox uses two axes.

The **main axis** follows `flex-direction`.

The **cross axis** is perpendicular to the main axis.

With:

```
flex-direction: row;
```

the main axis follows the row direction.

With:

```
flex-direction: column;
```

the main axis follows the column direction.

`flex-wrap: wrap` allows items to continue onto additional flex lines when the available space becomes too small.

Reverse directions can change visual order, but they do not rewrite the HTML source order. Visual order should not contradict reading and keyboard order.

10.3 Flex Basis, Growth, Shrinkage, Gaps, Alignment, Auto Margins, and Common Patterns

Flex item sizing is influenced by `flex-basis`, `flex-grow`, and `flex-shrink`.

```
.card {
   flex-basis: 17rem;
   flex-grow: 1;
   flex-shrink: 1;
}
```

`flex-basis` provides an initial main-axis size basis.
`flex-grow` controls participation in distributing positive free space.
`flex-shrink` controls participation when items need more space than the flex line provides.

The final size also depends on content and minimum-size constraints.

`gap` creates space between flex items and flex lines:

```
.card-list {
   display: flex;
   flex-wrap: wrap;
   gap: 1rem;
}
```

`justify-content` distributes items along the main axis when free space exists.
`align-items` controls cross-axis alignment.
`align-self` changes the cross-axis alignment of one item.

Auto margins can absorb free space. In a column flex card, `margin-top: auto` can move an action toward the end without giving the card a fixed height.

10.4 Grid Containers, Lines, Tracks, Cells, Areas, and Explicit and Implicit Grids

CSS Grid is designed for two-dimensional relationships involving rows and columns.

```
.panel {
   display: grid;
   grid-template-columns: 1fr 2fr;
}
```

A grid contains **lines**, **tracks**, **cells**, and **areas**.
A track is the space between two adjacent grid lines. Column tracks form columns and row tracks form rows.
A cell is one row-column intersection.
A grid area is a rectangular region covering one or more cells.
Named areas can describe layout structure:

```
.card {
   grid-template-areas:
      "date title"
      "date details";
}
```

The **explicit grid** is created by template declarations such as `grid-template-columns` and `grid-template-rows`.

The browser can create **implicit tracks** when content needs grid space outside that explicit structure.
Unexpected implicit tracks can reveal a placement error, but implicit tracks are also a normal part of Grid behaviour.

10.5 fr, repeat(), minmax(), Automatic Placement, and Introductory Responsive Grids

The `fr` unit represents a flexible fraction of available grid space.

```
grid-template-columns: 1fr 1fr 1fr;
```

creates three flexible tracks with equal flex factors.
`repeat()` reduces repeated syntax:

```
grid-template-columns: repeat(3, 1fr);
```

`minmax()` defines a track range:

```
grid-template-columns:
   repeat(3, minmax(12rem, 1fr));
```

Each track has a minimum of `12rem` and can grow flexibly.

An intrinsic responsive pattern is:

```
grid-template-columns:
    repeat(
       auto-fit,
       minmax(min(100%, 16rem), 1fr)
    );
```

This allows the number of tracks to adapt to available width without requiring a fixed device category.

Grid also has an automatic-placement algorithm. Unpositioned items are placed into available cells in source order under the normal auto-placement rules.

Intrinsic layout can solve many responsive problems before media queries are needed.

10.6 Choosing Normal Flow, Flexbox, Grid, and Introductory Positioning

Normal flow is appropriate when ordinary document order already describes the layout.

Flexbox is a strong choice when the main relationship is one-dimensional, such as a navigation row or action group.

Grid is a strong choice when rows and columns need to work together as a system.

These methods can coexist in one page.

Positioning has different purposes.

`position: relative` keeps an element in normal flow and can establish a positioning reference for descendants.

`position: absolute` removes an element from normal flow and positions it against an appropriate containing block.

`position: sticky` keeps an element in flow but can hold it near an inset edge during scrolling.

`position: fixed` removes an element from flow and commonly positions it relative to the viewport.

`z-index` participates in stacking order. A larger number does not automatically place an element above everything because stacking contexts create local layering environments.

Absolute and fixed positioning are not good substitutes for ordinary document layout because removed elements no longer reserve their normal flow space.

The main principle is to choose the least complicated layout system that accurately describes the relationship among the content.

Responsive Web Design

A webpage does not have one permanent size. The same document can appear in a narrow browser window, a large desktop window, split-screen mode, a phone in either orientation, or a zoomed browser.

Responsive design allows content, media, spacing, and layout relationships to adapt to the space and capabilities available.

11.1 Viewports, Mobile Browser Behaviour, Mobile-First Design, and Responsive versus Adaptive Layouts

The **screen** is the physical display. The **viewport** is the area through which the browser presents the document. They are not always the same size. CSS layout responds to the CSS viewport rather than to a marketing device name.

CSS pixel is also not necessarily one physical hardware pixel. Device pixel density, zoom, and browser behaviour affect the relationship.

Responsive HTML commonly includes:

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

This gives mobile browsers the expected initial layout relationship. It does not make rigid CSS responsive by itself.

A **mobile-first** approach begins with a useful narrow-layout base and adds enhancements when more space becomes available.

This does not mean that the design is only for phones. It means the base styles work under constrained space without needing a large layout first.

The words **responsive** and **adaptive** are used differently by different authors. A useful distinction is that responsive layouts change fluidly or conditionally, while adaptive layouts often switch among a smaller set of more distinct arrangements.

11.2 Fluid Containers, Flexible Media, Intrinsic Design, and Content-Based Breakpoints

A **fluid container** changes with available space:

```
.page-shell {
   width: 92%;
   max-width: 72rem;
   margin-inline: auto;
}
```

The percentage allows shrinkage. The maximum prevents unlimited expansion.

Flexible images can use:

```
img {
   max-width: 100%;
   height: auto;
}
```

This allows an image to shrink when its container becomes narrower while preserving its proportions.

An **intrinsic layout** lets content and available space influence the arrangement. The auto-fitting Grid pattern from Chapter 10 is an example.

A **content-based breakpoint** is chosen because the content relationship needs to change, not because a particular device name has been reached.

For example, a two-column layout may become useful only when both columns can keep adequate width. The breakpoint should reflect that content requirement.

11.3 Media Queries, Range Syntax, Orientation, Pointer, Hover, and Short Screens

A media query applies CSS only when a condition matches.

```
@media (width >= 52rem) {
   .content-layout {
      grid-template-columns: 2fr
1fr;
   }
}
```

Modern range syntax can use comparisons such as `>=`, `<`, and bounded ranges.

Traditional syntax such as `min-width` and `max-width` is also common.

Width is not the only possible condition. A page can also react to viewport height:

```
@media (height < 38rem) {
   .sticky-panel {
      position: static;
   }
}
```

Orientation queries describe viewport shape, not a particular device type.

Pointer and hover queries describe input capabilities. For example, hover styling can be limited to environments where the primary input can conveniently hover.

Essential information should not depend on hover because touch, keyboard, voice, and other input methods may not provide it.

11.4 Fluid Typography and Spacing with Relative Units and CSS Functions

Responsive values do not always need discrete breakpoints.

`clamp()` can create a bounded fluid value:

```
h1 {
   font-size:
      clamp(2rem, calc(1.5rem +
2vw), 3.25rem);
}
```

The result never falls below the minimum or rises above the maximum, while the preferred value can change between them.

This can be useful for selected headings, large spacing values, or layout gaps.

Ordinary body text does not need to depend only on viewport width. A stable root-relative size often produces more predictable reading behaviour.

Relative units such as `rem` also respect the relationship to the root font size, which can reflect user preferences.

A media query is useful when a layout relationship changes. A fluid function is useful when the same property can change continuously within limits.

11.5 Responsive Navigation, Cards, Tables, Forms, Videos, and Embeds

Responsive design applies to individual components as well as the page wrapper.

Navigation labels may wrap or move from stacked to horizontal layouts as space changes. Hiding destinations simply because they do not fit is not a complete responsive strategy.

Cards can use intrinsic Grid sizing and content-driven height. Fixed heights are risky because headings and descriptions can grow.

Wide data tables can use local horizontal scrolling when their two-dimensional structure cannot reflow without losing meaning.

Forms can remain one column in narrow space and become multi-column at wider content-based breakpoints while preserving source and focus order.

Native video can scale with:

```
video {
   width: 100%;
   height: auto;
}
```

An iframe can use a flexible width and `aspect-ratio`, but the embedded document must also be responsive inside the frame.

Responsive presentation does not remove semantic or accessibility requirements. Captions, labels, headings, and fallback links still matter at every size.

11.6 Testing Narrow and Wide Viewports, Zoom, Reflow, Targets, Source Order, and Content Extremes

Responsive behaviour must be observed across a range of conditions.

Useful conditions include:

- narrow, middle, and wide viewport sizes;
- widths just below and just above breakpoints;
- full-page zoom;
- larger default fonts;
- long headings and labels;
- short viewport heights;
- keyboard focus;
- local table scrolling;
- media and iframe states.

A page that works at one phone preset and one desktop preset can still fail between them.

Zoom changes the effective CSS viewport. Reflow evaluation therefore depends on the actual reported CSS width rather than only the physical monitor size.

Source order should remain logical even when CSS changes visual arrangement. Flexbox and Grid should not create a visual order that conflicts with reading or focus order.

Real content extremes are important because placeholder text is usually shorter and easier than production content.

The main principle is that responsive design is successful when content and functions remain understandable and usable as conditions change.

Accessibility, Testing, Performance Basics, and Publishing

A website is not complete when it looks correct. It also needs understandable structure, usable interaction, reliable testing, reasonable performance, and correct public delivery.

This chapter brings together principles from the rest of the book and explains how they are evaluated.

12.1 Accessibility Principles, Semantic HTML, Landmarks, Headings, Names, and Descriptions

WCAG organises accessibility under four broad principles:

```
Perceivable
Operable
Understandable
Robust
```

These are commonly shortened to **POUR**.

Perceivable means information can be perceived in appropriate forms, such as text alternatives for meaningful images and sufficient contrast.

Operable means interface functions can be used, including through a keyboard where appropriate.

Understandable means information, labels, instructions, and interaction are clear and reasonably predictable.

Robust means markup communicates roles, names, states, values, and relationships in ways browsers and assistive technologies can interpret.

Semantic HTML supports this work. Elements such as `main`, `nav`, `header`, `aside`, and `footer` can create meaningful regions.

Headings should describe content hierarchy.

An **accessible name** identifies an element, such as the text of a button or a form label.

An **accessible description** adds supporting information. It does not replace the name.

Native HTML usually provides better built-in semantics and behaviour than generic elements with added ARIA roles.

12.2 Keyboard Navigation, Focus Order, Visible Focus, and Component Auditing

Keyboard accessibility matters because not every user operates a website with a mouse or touch screen.

Sequential focus order should follow a logical task and document sequence.

Positive `tabindex` values can create a focus order that conflicts with source order and are generally unsuitable for page-wide navigation control.

Focus also needs to be visible:

```
:focus-visible {
   outline: 3px solid #111;
   outline-offset: 3px;
}
```

A focus indicator can still fail if it is hidden behind a sticky header, clipped by overflow, or too similar to the background.

Native links and buttons already have established keyboard behaviour. A styled `div` with `tabindex="0"` does not automatically become a real button.

Component auditing considers the relationships inside forms, tables, images, media, navigation, and scrollable regions rather than only the appearance of the page.

12.3 Contrast, Colour Independence, Zoom, Reflow, Orientation, Target Size, and User Preferences

Text needs sufficient contrast against its background. WCAG 2.2 Level AA uses a minimum ratio of `4.5:1` for ordinary text and `3:1` for sufficiently large text.

Important non-text visual information can also have contrast requirements.

Colour should not be the only way to communicate meaning. A status such as “Places available” should be written in words rather than represented only by green colour.

Text enlargement and reflow test different concerns. Content should remain usable when text becomes larger, and vertically scrolling content should normally adapt to a narrow effective CSS width without forcing whole-page two-dimensional scrolling, subject to defined exceptions.

Orientation changes should not make ordinary information unavailable.

WCAG 2.2 includes a Level AA minimum target-size criterion of `24 × 24` CSS pixels with defined exceptions.

User preferences can also affect presentation. Media features such as `prefers-contrast` and `forced-colors` allow CSS to respond to some accessibility settings.

These enhancements work best when the base design is already complete and usable.

12.4 HTML and CSS Validation, Manual Checks, Automated Tools, Screen Readers, and Cross-Browser Testing

Different tests provide different kinds of evidence. HTML validation can find invalid nesting, duplicate IDs, obsolete markup, and other source problems. CSS validation can identify syntax errors and invalid property values.

Validation does not prove accessibility. A declaration such as:

```
:focus {
   outline: none;
}
```

can be syntactically valid while creating a serious usability problem.

Automated accessibility tools can find some missing names, contrast problems, invalid ARIA relationships, and structural patterns. They cannot reliably judge every heading, alternative text, caption, instruction, or task sequence.

Manual testing is therefore necessary for questions that require human judgement or real interaction.

Browser accessibility tools can expose computed roles, names, descriptions, and states. This is useful evidence, but it is not identical to actual screen-reader output.

Screen-reader testing should record the real browser, operating system, screen reader, version, and observed behaviour.

Cross-browser testing should focus on the environments that matter to the project and on features with real compatibility or interaction risk.

12.5 Image, Font, Stylesheet, and Layout-Shift Performance Basics

Performance begins with understanding which resources a page loads and which ones matter to the user experience.

Image performance depends on file dimensions, format, compression, candidate selection, and when the resource is requested.

Responsive images allow the browser to select among candidates. Smaller files can reduce transfer size, but smaller bytes do not automatically prove better visual quality or better performance metrics.

Web fonts add network requests. A system font stack can avoid that cost. When custom fonts are necessary, the number of faces, format, fallback behaviour, and licensing all matter.

Stylesheets should remain valid and reasonably organised. Complex optimisation systems are not automatically useful for small projects.

Current Core Web Vitals include:

- **LCP**, which relates to the rendering time of the largest relevant content in the initial view;
- **INP**, which relates to responsiveness to user interaction;
- **CLS**, which measures unexpected layout movement.

Intrinsic image dimensions can reduce one cause of layout movement, but they do not prove a particular CLS result.

Performance evidence can come from static file inventories, browser network panels, performance traces, synthetic lab tools, and real-user field data. Each type has different limits.

12.6 Hosting, Domains, HTTPS, Deployment, 404 Pages, Release Checks, and README Documentation

A local website becomes public through several connected systems.

Hosting provides infrastructure that serves resources.

A **domain name** gives a human-readable address.

DNS connects domain names with network destinations and services.

HTTP carries requests and responses.

HTTPS protects HTTP transport with TLS.

These are related but separate layers.

Deployment also changes how paths behave. A project that works locally can fail publicly because of case-sensitive filenames, wrong publish folders, different base paths, redirects, or missing resources.

A custom `404.html` file can provide a useful error page, but the server must also return the real HTTP status `404 Not Found`. A page that looks like an error but returns `200 OK` is a soft 404.

HTTPS confirms protected transport and certificate-based server identity. It does not prove that the website is truthful, accessible, or secure at the application level.

A README can document the project purpose, file structure, local environment, deployment state, licences, and known limitations. Public documentation should never contain passwords, API keys, private keys, or other secrets.

A release should be identifiable so that testing evidence refers to a known build. Public testing remains necessary after deployment because server configuration, MIME types, redirects, caching, HTTPS, and real resource paths can differ from the local environment.

The central idea of the final chapter is evidence. A professional website is evaluated through several kinds of checks, and each check should be understood according to what it can and cannot prove.

COGNITIA
EUROPEAN INSTITUTE OF DIGITAL SKILLS

About Cognitia

Cognitia is an online learning platform focused on practical, accessible education for people who want to build useful skills for today's digital world.

Our courses, ebooks, and learning resources cover topics such as UX/UI design, web development, artificial intelligence, and other modern digital skills. Each resource is created to explain complex subjects clearly and help beginners learn step by step through practical examples.

Learn more at: **www.ceids.top**

COGNITIA

EUROPEAN INSTITUTE OF DIGITAL SKILLS

This book is part of the educational materials created for **Cognitia** and is based on the **Cognitia HTML & CSS for Beginners** course.

www.ceids.top