ChatGPT

HTML5: 100 concetti fondamentali

Sintassi e struttura base del documento HTML5

- Il doctype HTML5 è semplice e universale: la prima riga del documento deve essere `<!DOCTYPE html>`, per indicare al browser di usare la modalità standard (evitando la modalità "quirks") [1].
- L'elemento radice `<html>` racchiude tutto il contenuto e include l'attributo `lang` per specificare la lingua principale della pagina, migliorando l'accessibilità e l'indicizzazione [2].
- La struttura di base comprende `<head>` (metadati non visibili, come titolo e meta tag) e `<body>` (contenuto visibile), entrambe figlie di `<html>` [3].
- Nel `<head>` vanno indicati elementi essenziali: ad esempio `<meta charset="UTF-8">` per dichiarare la codifica (UTF-8 è lo standard in HTML5) [4], e `<title>` per definire il titolo della pagina (visualizzato nella scheda del browser e nei risultati dei motori di ricerca) [5].
- Per siti responsive è fondamentale includere `<meta name="viewport" content="width=device-width, initial-scale=1.0">`: questo meta tag assicura che la pagina si adatti alla larghezza del dispositivo, facilitando la visualizzazione su mobile [6].
- HTML5 è più permissivo sulle chiusure di tag rispetto a XHTML: molti tag di chiusura sono opzionali (ad esempio, si possono omettere `</p>` o `</li>` perché la chiusura è implicita) e i tag vuoti non richiedono la barra finale (`<br>` è valido come `<br/>`) [7].
- I tag e gli attributi HTML5 non sono case-sensitive (per convenzione si usano minuscole). Allo stesso modo, le virgolette intorno ai valori degli attributi sono consigliate, ma HTML5 li accetta anche senza virgolette se non contengono spazi (es: `content=value`).
- HTML5 introduce una sintassi semplificata rispetto alle versioni precedenti: non richiede riferimenti a DTD o attributi come `type` per gli script e i fogli di stile (sono presupposti di default), e supporta direttamente caratteristiche come video e audio senza plugin esterni.
- È **retrocompatibile**: pagine HTML4/XHTML continuano a funzionare in HTML5 (i browser ignorano i tag sconosciuti), sebbene molti tag e attributi obsoleti siano deprecati. Ad esempio, elementi presentazionali come `<font>` o `<center>` sono stati eliminati dallo standard HTML5: vanno sostituiti con CSS per la grafica e con elementi semantici appropriati per la struttura.
- Ogni pagina HTML5 dovrebbe essere costruita in **modalità standard** e valida: mantenere un markup ben formato e usare un validatore (come il **W3C Markup Validator**) aiuta a individuare errori di sintassi e garantisce un comportamento coerente su diversi browser.
- L'**ordine degli elementi** è importante: il doctype in cima, seguito da `<html>` con lingua, poi `<head>` (con meta, titolo, link a CSS/script) e infine `<body>` con il contenuto. HTML5 permette di omettere la chiusura di `</head>` e `</body>` (il parser la inserisce automaticamente), ma è buona pratica includerli per chiarezza.
- Il **commento** in HTML è `<!-- ... -->`. I commenti non vengono visualizzati e servono a documentare il codice; assicurarsi di chiuderli correttamente per non mandare fuori uso porzioni di markup.

Tag principali e semantici (header, section, article, footer...)

- HTML5 ha introdotto molti **tag semantici** che danno significato al layout della pagina, in sostituzione dei generici `<div>`: ciò migliora l'organizzazione del codice e aiuta browser e motori di ricerca a capire la struttura del contenuto [8].
- `<header>` definisce la **testata di una pagina o sezione**: di solito contiene il titolo principale, eventuali sottotitoli, logo o menu di navigazione introduttivi [9].
- `<nav>` identifica un blocco di **collegamenti di navigazione**: ad esempio il menu principale di un sito o i link di paginazione. Facilita agli utenti (e screen reader) l'individuazione delle sezioni di menu [10].
- `<main>` rappresenta il **contenuto principale** della pagina, quello più significativo (andrebbe usato una sola volta per pagina). Ad esempio, il testo dell'articolo in una pagina di blog è racchiuso in `<main>` [11].
- `<section>` definisce una **sezione tematica** del documento, tipicamente con un proprio intestazione (`<h2>` etc.). Si usa per raggruppare contenuti affini (es: capitoli, sezioni di un capitolo) all'interno di articoli o pagine [12].
- `<article>` indica un **contenuto autonomo e riutilizzabile**: ad esempio un articolo di blog, una notizia, un post di forum o un widget. Un `<article>` ha senso di per sé e potrebbe essere distribuito indipendentemente (feed, aggregatori) [13].
- `<aside>` rappresenta un **contenuto complementare o di contorno** rispetto a quello principale. Tipicamente è usato per barre laterali, riquadri informativi, note o elementi correlati che affiancano il contenuto centrale [14].
- `<footer>` definisce il **piè di pagina** di una sezione o pagina: può contenere informazioni di chiusura come note, link correlati, copyright, informazioni di contatto o navigazione di fondo [15].
- `<figure>` e `<figcaption>` servono insieme per inserire contenuti illustrativi con didascalia. `<figure>` racchiude un elemento (un'immagine, diagramma, codice, tabella, ecc.) e `<figcaption>` fornisce la **didascalia descrittiva** associata [16].
- `<details>` definisce un **contenuto espandibile/collassabile** che l'utente può mostrare o nascondere a richiesta. Si usa in coppia con `<summary>` che ne indica l'intestazione o titolo visibile sempre [17] [18]. Ad esempio, FAQ in cui cliccando sul `<summary>` (domanda) si espande il `<details>` (risposta).
- `<summary>` è l'elemento che **titola un blocco** `<details>`: il suo contenuto è sempre visibile e cliccabile, e controlla l'apertura/chiusura dei dettagli associati [18].
- `<mark>` evidenzia un **testo di rilievo o evidenziato** all'interno del contenuto, generalmente mostrato con uno sfondo giallo dai browser. Si usa per sottolineare parti importanti o termini di ricerca all'interno di un risultato [19].
- `<time>` rappresenta un **valore temporale o data** in un formato leggibile sia dagli umani sia dalle macchine. Si può usare l'attributo `datetime` per specificare il valore in formato standard (ISO 8601), mentre il contenuto visibile può essere una data formattata o testo naturale [20].
- Oltre a questi, HTML5 comprende altri tag semantici come `<address>` (informazioni di contatto), `<cite>` (titoli di opere citate), `<dfn>` (definizioni), `<dialog>` (finestre di dialogo modali) e così via. L'uso appropriato di ciascun elemento semantico al posto di anonimi `<div>` migliora la **leggibilità del codice**, l'**accessibilità** (es. gli screen reader riconoscono sezioni, navigazione, ecc.) e la **SEO**, poiché fornisce un significato esplicito ai diversi blocchi di contenuto.

Gestione di immagini, video e altri media

- `<img>` è il tag per inserire **immagini**. È importante fornire sempre l'attributo obbligatorio `alt` con una descrizione testuale dell'immagine (o alt="" se decorativa) per accessibilità e SEO.

Si possono specificare dimensioni con `width` / `height` (in pixel) per riservare lo spazio ed evitare spostamenti di layout durante il caricamento [21].

- HTML5 introduce meccanismi per **immagini responsive**: l'attributo `srcset` (insieme a `sizes`) permette al browser di scegliere automaticamente l'immagine più adatta tra varie versioni (es. diverse risoluzioni per schermi retina vs normali), mentre il tag `<picture>` consente di definire sorgenti alternative con media queries (per mostrare immagini diverse in base alle dimensioni dello schermo o densità pixel).
- `<video>` permette di incorporare **video** nativamente, senza plugin come Flash. Supporta attributi utili: `controls` (mostra i controlli di riproduzione integrati: play/pausa, volume, ecc.), `autoplay` (avvio automatico, richiede spesso `muted` per policy dei browser), `loop` (ricomincia il video automaticamente a fine riproduzione) e `poster` (URL di un'immagine da mostrare come anteprima prima che il video parta) [22]. È buona pratica definire anche `width` e `height` per fissare le dimensioni del player video.
- `<audio>` inserisce un **contenuto audio** (musica, suoni) con controlli nativi. Analogamente al video, ha `controls` per mostrare ad es. play/pausa e barra di avanzamento, `autoplay`, `loop` e `muted`. Poiché l'audio non ha componente visiva, non esiste un attributo poster (l'elemento audio rimane invisibile se `controls` non è impostato) [23].
- Sia `<video>` che `<audio>` possono includere all'interno uno o più tag `<source>` per fornire più formati dello stesso contenuto. Ogni `<source>` indica un file (`src="file.mp4"` etc.) e tipologia (`type="video/mp4"` ad esempio) [24] [25]. Il browser sceglierà automaticamente il primo formato che supporta, garantendo compatibilità cross-browser (es: MP4, WebM e Ogg per i video) [26].
- Per **sottotitoli e didascalie** nei media, HTML5 offre `<track>` all'interno di video/audio. Questo tag permette di aggiungere testi sincronizzati, come sottotitoli (`kind="subtitles"` con `srclang` per la lingua e magari `label`), sottotitoli per non udenti (`kind="captions"`) o capitoli e metadati. I sottotitoli migliorano l'accessibilità e possono essere attivati dall'utente nei controlli del player [27].
- `<canvas>` definisce un'area di disegno 2D nello schermo (di dimensioni specificate con width/height o CSS). Non mostra contenuto di per sé, ma tramite JavaScript si ottiene un contesto grafico (metodo `getContext('2d')`) su cui disegnare forme, immagini e testo in tempo reale. È ideale per **grafica dinamica**, giochi, grafici o animazioni generati via script [28].
- È possibile includere **grafica vettoriale SVG** direttamente nell'HTML5 usando il tag `<svg>` (e relativi sottotag `<circle>`, `<rect>`, `<path>`, etc.): questo consente di disegnare figure scalabili (che non perdono qualità al variare della risoluzione) e interattive, senza bisogno di immagini esterne. In alternativa, si può usare `<img src="disegno.svg">` o incorporare SVG con `<object>` / `<embed>`.
- Per **contenuti esterni** come mappe, video di piattaforme (YouTube/Vimeo) o altri siti, si utilizza l'elemento `<iframe>` che embedda una pagina esterna all'interno di un riquadro. L'iframe supporta attributi come `src` (URL del contenuto da visualizzare), `width` / `height` (dimensioni) e opzioni di sicurezza come `sandbox` (per limitare l'esecuzione di script) e `allow` (per concedere permessi specifici, es. fullscreen, accelerometro, ecc.).
- HTML5 elimina la necessità di plugin proprietari per i media: audio e video si riproducono nativamente. Ad esempio, non servono più oggetti Flash per i video – basta `<video>` –, il che risolve molti problemi di sicurezza e compatibilità mobile [29]. Per i formati non supportati da HTML5 (es. vecchi codec), si possono comunque fornire alternative: usare più `<source>` con formati diversi o, in casi estremi, un link di download/video esterno come fallback tra i tag `<video>...</video>` (che verrà mostrato se il browser non supporta l'elemento video) [30] [31].
- **Responsive e performance media**: usare attributi come `preload` su video/audio per controllare il caricamento (es.: `preload="none"` per non scaricare nulla finché l'utente non

avvia, risparmiando banda, oppure `"metadata"` per caricare solo durata e anteprima). Inoltre, specificare dimensioni fisse o massime per media aiuta la resa responsive (video e canvas possono essere adattati via CSS), mentre per immagini responsive conviene usare `srcset` / `sizes` o `<picture>` come detto, così il browser evita di caricare immagini troppo grandi su schermi piccoli.

- **Accessibilità dei media**: fornire alternative testuali è cruciale. Per audio, includere una trascrizione del parlato; per video, sottotitoli tramite `<track>` o un testo descrittivo della scena. In caso di `<canvas>`, se viene usato per mostrare informazioni importanti, prevedere un fallback testuale (inserendo testo tra `<canvas>...</canvas>` che appare se il browser non supporta Canvas).

Accessibilità (tag semantici, ARIA, buone pratiche)

- **Testi alternativi**: assicurarsi che ogni contenuto non testuale abbia un'alternativa testuale. Ogni `<img>` deve avere un attributo `alt` descrittivo (se l'immagine è puramente decorativa, alt="" in modo che lo screen reader la ignori) [32] . Analogamente, per video e audio bisogna fornire sottotitoli o trascrizioni, e per elementi canvas o SVG complessi si deve prevedere una descrizione testuale (magari nascosta).
- **Struttura semantica e intestazioni**: utilizzare i tag di intestazione `<h1>`–`<h6>` in ordine logico senza saltare livelli (es.: non passare da h1 a h3 senza un h2 intermedio). Una pagina dovrebbe avere un solo `<h1>` che rappresenta il titolo principale, seguito da eventuali `<h2>` per sezioni, `<h3>` per sottosezioni e così via. Questo crea una gerarchia che gli screen reader usano per **navigare la pagina** (permette ad esempio di saltare da sezione a sezione) [32] .
- **Form accessibili**: ogni campo di input deve essere associato a un'etichetta testuale. Usa sempre `<label>` per indicare la funzione dell'input (collegando l'etichetta al campo tramite l'attributo `for` e un id coincidente, oppure racchiudendo l'input nel label). Ad esempio: `<label for="email">Email: <input id="email" type="email"></label>`. Questo assicura che i lettori di schermo annuncino il nome del campo quando l'utente vi entra. In mancanza di `<label>`, usare attributi ARIA come `aria-label="Descrizione"` o `aria-labelledby="id_elemento_etichetta"` per fornire un nome accessibile al controllo.
- **Navigazione da tastiera**: garantire che tutti gli elementi interattivi siano accessibili tramite **tastiera** (non solo mouse). Ciò significa che link, bottoni, campi di input devono essere raggiungibili con il tasto Tab e attivabili con Invio/Spazio. Per elementi non nativamente focusable (es. un `<div>` usato come pulsante via JavaScript), bisogna aggiungere `tabindex="0"` per inserirlo nell'ordine di tabulazione e gestire gli eventi keypress. Inoltre, non rimuovere mai via CSS l'indicatore di focus (outline) senza fornire un'alternativa visiva equivalente: gli utenti che navigano da tastiera devono vedere quale elemento ha il focus attualmente.
- **Uso di ARIA**: *WAI-ARIA* (Web Accessibility Initiative – Accessible Rich Internet Applications) è un set di attributi progettati per migliorare l'accessibilità di componenti non nativamente accessibili. ARIA consente di definire ruoli, stati e proprietà su elementi HTML. Ad esempio, `role="button"` su un `<div>` comunica ai dispositivi assistivi che quell'elemento si comporta come un bottone, `aria-pressed="false"` può indicare stato non premuto di un pulsante toggle, `aria-live="polite"` fa sì che un'area venga annunciata quando cambia (utile per notifiche dinamiche), e così via [33] [34] .
- **Prima la semantica, poi ARIA**: regola fondamentale – *se puoi usare un elemento HTML semantico o nativo, fallo, invece di usare ARIA!* ARIA va impiegato solo quando **non esiste un equivalente HTML** per ottenere quel comportamento o informazione [35] [36] . Esempio: meglio usare un vero `<button>` invece di un link con `role="button"`; un `<nav>` invece di un div con `role="navigation"`. Gli elementi HTML5 come `<main>`, `<header>`, `<footer>`, `<nav>`

hanno già ruoli impliciti che i reader riconoscono, quindi non serve aggiungere ARIA in quei casi (non mettere `role="main"` dentro `<main>`, ecc.).

- **Ruoli ARIA per regioni**: se per qualche motivo non si possono usare i tag HTML5 semantici, ARIA offre ruoli di *landmark* per marcare regioni della pagina: ad esempio `role="navigation"` per la barra di navigazione, `role="main"` per il contenuto principale, `role="banner"` per l'header, `role="contentinfo"` per il footer. In HTML5 questi landmark sono già coperti dai tag corrispondenti, ma potresti incontrarli in codice legacy o usarli per retrocompatibilità con browser vecchi.
- **Elementi interattivi e stati**: ARIA permette di definire stati e proprietà accessibili. Ad esempio, `aria-expanded="false"` / `true` su un menu a fisarmonica indica se la sezione è aperta o chiusa (lo screen reader annuncerà "espanso"/"compresso"), `aria-current="page"` sul link attuale del menu segnala la pagina corrente, `aria-disabled="true"` rende un controllo percepito come disabilitato. Usali per aggiornare dinamicamente l'esperienza utente: es. pulsante *Mi piace* che cambia `aria-pressed` quando cliccato. Ricorda di aggiornare questi attributi via script quando cambia lo stato dell'UI.
- **Contenuti nascosti**: per nascondere un elemento ai soli screen reader mantenendolo visivo, *non* usare `display:none` (lo rimuove sia visualmente che dall'accessibility tree). Invece, potresti usare tecniche CSS di *visually hidden* (lasciandolo fuori schermo). Viceversa, per nascondere un elemento visivamente **e** agli screen reader, HTML5 offre l'attributo `hidden` (equivalente ad aria-hidden="true" e display:none insieme). Ad esempio: `<div hidden>Testo nascosto</div>` non verrà letto né mostrato.
- **Contrasto e colori**: anche se gestito via CSS, è parte dell'accessibilità web. Assicurati che il contrasto tra testo e sfondo sia sufficiente (WCAG consiglia un contrasto di almeno 4.5:1 per testo normale). Non affidarti solo al colore per trasmettere informazioni (es.: non dire "i campi in rosso sono obbligatori" senza indicarlo anche nel testo o con icone/testo aggiuntivo). Usa icone con `aria-label` o testo nascosto per trasmettere lo stesso messaggio ai non vedenti.
- **Sottotitoli, didascalie e traduzioni**: per contenuti multimediali, fornisci sempre alternative: sottotitoli sincronizzati per i video (o almeno un riassunto se sottotitoli completi non sono possibili) [37] , trascrizioni per i podcast/audio, audiodescrizioni per video se contengono informazioni visive importanti. Questo non solo aiuta utenti con disabilità sensoriali, ma spesso è richiesto da standard come WCAG.
- **Skip links**: implementa collegamenti di salto, visibili solo a chi naviga da tastiera, che permettano di saltare direttamente al contenuto principale. Di solito è un link all'inizio della pagina tipo `<a href="#main-content" class="visually-hidden focusable">Salta al contenuto principale</a>`. Quando si preme Tab all'inizio, appare e consente di bypassare menu e header ripetitivi, migliorando l'usabilità per chi usa tecnologie assistive o solo la tastiera.
- **Testing**: prova sempre il tuo sito con strumenti assistivi. Usa uno screen reader (NVDA, JAWS su Windows, VoiceOver su Mac), prova a navigare solo con la tastiera, e utilizza validatori automatici (come *WAVE* o *AXE*) per individuare problemi comuni. L'accessibilità è più efficace se considerata sin dall'inizio, non come riparazione a fine sviluppo.

SEO: ottimizzazione semantica, struttura, meta tag e attributi

- **Title tag (titolo della pagina)**: è uno dei fattori SEO on-page più importanti. Va incluso nell'head e contiene un breve testo che descrive la pagina. È ciò che appare come titolo cliccabile nei risultati di Google e nella barra del browser [38] . Deve essere unico per ogni pagina, conciso (idealmente sotto ~60 caratteri) e rilevante, così da invogliare al clic e indicare chiaramente l'argomento al motore di ricerca.
- **Meta description**: è un meta tag facoltativo, ma altamente raccomandato, nell'head: `<meta name="description" content="Descrizione del contenuto...">`. Fornisce una breve

sintesi della pagina (circa 150-160 caratteri). I motori spesso la mostrano sotto il titolo nei risultati (snippet) [39] . Non influisce direttamente sul ranking [40] , ma una description ben scritta può aumentare il **CTR** (più utenti cliccano se lo snippet è accattivante), influenzando indirettamente il posizionamento.

- **Intestazioni (heading tags)**: Usa correttamente `<h1>...<h6>` per strutturare il contenuto. `<h1>` dovrebbe contenere il tema principale o parola chiave principale della pagina (preferibilmente una sola volta). Le sottosezioni dovrebbero avere `<h2>` e via dicendo. I search engine utilizzano queste gerarchie per comprendere l'organizzazione e i topic della pagina [41] . Ad esempio, un `<h2>` potrebbe fungere da sottotitolo contenente parole chiave correlate. Una struttura di heading ben formata migliora sia l'user experience sia la **comprensibilità SEO** del testo.
- **Tag semantici HTML5**: l'uso di elementi come `<header>`, `<nav>`, `<article>`, `<footer>`, ecc. non solo migliora l'accessibilità, ma *aiuta i motori di ricerca a capire meglio le parti della pagina* [42] . Ad esempio, Google riconosce facilmente il contenuto principale se racchiuso in `<main>` o `<article>`, distingue i menu di navigazione in `<nav>`, e sa che `<footer>` contiene info aggiuntive. Ciò può influire su come viene estratto il contenuto per i risultati (ad es. Google potrebbe ignorare il menu e focalizzarsi sul testo dell'articolo per determinare la rilevanza).
- **Attributi ALT per immagini**: dal punto di vista SEO, il testo alternativo delle immagini è fondamentale. Poiché i motori non "vedono" le immagini, si basano sull'attributo `alt` per capire cosa rappresentano. Un alt ben scritto (descrittivo e conciso, possibilmente con una keyword se pertinente) aiuta a far comparire l'immagine nei risultati di Google Immagini e aggiunge contesto alla pagina [43] [44] . Non riempire l'alt di parole chiave (keyword stuffing): deve essere utile e inerente all'immagine.
- **URL e struttura dei link**: utilizzare URL *parlanti* e semplici (ad es. `miosito.it/blog/come-fare-SEO` anziché parametri incomprensibili). In HTML, assicurati che i link interni siano testuali e descrittivi (il testo del link dovrebbe indicare la destinazione, es. *"Guida SEO"* invece di *"clicca qui"*). Ciò migliora la comprensione da parte di Google e l'esperienza utente. Inoltre, aggiungi collegamenti interni tra pagine correlate con anchor text significativi per distribuire l'autorevolezza e far capire le relazioni tra contenuti.
- **Meta robots**: questo meta tag (nel head) controlla l'indicizzazione di una pagina da parte dei crawler. Esempio: `<meta name="robots" content="noindex, nofollow">` dice ai motori di **non indicizzare** la pagina e di **non seguire** i link in essa [45] . Si usa tipicamente per pagine duplicate, pagine in staging, risultati di ricerca interni o contenuti che vuoi escludere dall'indice. Puoi specificare direttive come `index` / `noindex`, `follow` / `nofollow`, e anche altre (es. `noarchive`, `nosnippet`). È uno strumento di controllo fine, da usare con cautela: ad esempio, un `noindex` erroneo su una pagina importante la escluderà completamente dai risultati.
- **Link canonical**: se la stessa risorsa è accessibile tramite più URL (ad es. versioni con e senza parametri, HTTP/HTTPS, paginazioni, o contenuti molto simili), usa `<link rel="canonical" href="URL-preferita">` nel head per indicare a Google qual è la versione canonica da indicizzare [46] . Questo aiuta a prevenire contenuti duplicati nei risultati e consolida il "peso" SEO su un unico URL [47] . Ad esempio, una scheda prodotto con parametri diversi per filtri dovrebbe canonizzare sull'URL base del prodotto. Google considera il canonical come un *hint* (suggerimento, non obbligo) ma in generale lo rispetta, aggregando ranking e backlink alla pagina canonica.
- **Ottimizzazione mobile**: con l'avvento del *mobile-first indexing*, la versione mobile del sito è quella primaria valutata da Google. Quindi assicurati che il tuo HTML sia **responsive**. Il già citato meta viewport è essenziale. Inoltre, evita elementi non supportati su mobile (Flash, plugin vari) e usa dimensioni relative o CSS flex/grid per adattare il layout. Per SEO, un sito mobile-friendly

ottiene migliori valutazioni (Google ha indicatori di *Page Experience* come Core Web Vitals che influenzano leggermente il ranking).

- **Dati strutturati (structured data)**: HTML5 permette di incorporare metadati semantici nel codice tramite microdata o JSON-LD. Puoi usare attributi *microdata* (itemscope, itemtype, itemprop) direttamente sugli elementi HTML o, metodo più moderno, includere nel `<head>` uno script JSON-LD con vocabolario Schema.org. I **dati strutturati** aiutano i motori a comprendere entità specifiche nella pagina (es. recensioni, eventi, ricette, prodotti) e possono abilitare i *rich snippet* (risultati arricchiti) nei risultati di ricerca. Ad esempio, segnando con Schema.org le stelle di rating e il numero di recensioni, Google potrebbe mostrare le ⭐ e il conteggio nelle SERP. Implementa markup strutturato attenendoti alle linee guida di Google (evita di marcare info non visibili o non pertinenti) e verifica con il **Rich Results Test** di Google.
- **Attributi link social (Open Graph e Twitter Cards)**: non influenzano il ranking, ma sono meta tag importanti per la *condivisione social*. `<meta property="og:title">`, `og:description`, `og:image` ecc. (protocollo Open Graph di Facebook, usato anche da LinkedIn) e `<meta name="twitter:...">` per Twitter definiscono come apparirà la preview della pagina quando condivisa sui social [48] [49]. Anche se non direttamente SEO, avere anteprime ottimizzate migliora il click-through quando i contenuti vengono scoperti tramite social, portando traffico che indirettamente può giovare.
- **Nofollow e link esterni**: in HTML, aggiungere `rel="nofollow"` su un link esterno indica ai motori di *non trasferire PageRank* a quella destinazione [50]. Google tratta i nofollow come *hint*, ma in generale non segue né considera quei link per il ranking. Usa nofollow (o i più specifici `rel="sponsored"` per link a pagamento e `rel="ugc"` per contenuti generati dagli utenti) nei collegamenti che non vuoi "consigliare" a Google – ad esempio link pubblicitari, nei commenti utenti, o in generale link a cui non vuoi essere associato per SEO. È buona pratica bilanciare: linka liberamente a risorse autorevoli (do-follow) per arricchire l'esperienza utente, ma usa nofollow se hai dubbi sulla qualità o natura promozionale del link.
- **Prestazioni e esperienza utente**: molti aspetti tecnici HTML incidono sulla SEO. Un codice pulito, senza errori, migliora la velocità di caricamento e la compatibilità cross-browser. La velocità di caricamento è un fattore di ranking minore ma esistente. Minimizza l'uso di HTML superfluo (evita nidificazioni inutili di div, commenti eccessivi, ecc.). Utilizza l'attributo `loading="lazy"` su immagini/iframe per differirne il caricamento finché non sono nei pressi della viewport [51], riducendo il tempo di caricamento iniziale. Includi le risorse CSS in `<head>` in modo che vengano caricate rapidamente e non blocchino il rendering (o usa l'attributo `rel="preload"` per caricare in anticipo risorse critiche). Una buona **struttura HTML** unita a best practice di performance migliora i Core Web Vitals (LCP, FID, CLS), che Google utilizza come segnali di esperienza utente.

Moduli HTML5: tipi di input, validazione, nuovi attributi

- HTML5 ha arricchito i form con **nuovi tipi di input** per migliorare l'esperienza utente e la semantica. Oltre al classico `type="text"`, abbiamo: `email`, `url`, `tel` (telefono), `number` (numerico), `range` (slider di intervallo), `date`, `time`, `month`, `week`, `datetime-local` (vari tipi per selezione di date/ore), `color` (selettore di colore), `search` (campo di ricerca) ecc. [52]. Questi tipi forniscono controlli specifici (ad es. tastiera mobile adattata, widget browser come calendari e color picker) e validazione integrata del formato.
- I campi `<input type="email">` e `<input type="url">` effettuano una **validazione automatica** sul formato inserito. Se l'utente digita qualcosa che non corrisponde a un indirizzo email valido (per `email`) o a un URL (per `url`), il form non verrà inviato e il browser mostrerà un messaggio di errore predefinito [53]. Inoltre, su dispositivi mobili, questi campi fanno comparire tastiere ottimizzate (ad esempio, per email appare il tasto "@" a portata di mano).

- `<input type="number">` accetta solo input numerici (interi o decimali). I browser spesso aggiungono piccole frecce su questo campo per aumentare/diminuire il valore. Puoi specificare gli attributi `min` e `max` per definire l'intervallo consentito e `step` per l'incremento (es. step="1" o "0.1"). Ad esempio: `<input type="number" name="quantita" min="1" max="10">` permette solo numeri tra 1 e 10 [54]. Se l'utente inserisce un numero fuori range, il campo risulterà non valido.
- I nuovi input per date e orari (`date`, `time`, `datetime-local`, `month`, `week`) forniscono **widget nativi** nei browser compatibili. Ad esempio, `<input type="date">` farà apparire un selettore di data (calendario) per scegliere giorno/mese/anno [55], mentre `<input type="datetime-local">` include anche orario (senza fuso orario). Anche questi supportano `min`, `max` e `step` (es. step="7" su date potrebbe saltare di 7 giorni, creando un calendario settimanale). In vecchi browser che non li supportano, degradano a normalissimi campi testo.
- **Attributo required**: HTML5 introduce `required` per indicare che un campo è obbligatorio. Se un input con required è vuoto al tentativo di invio, il browser blocca l'invio e segnala che il campo va compilato [56]. Ciò evita di dover scrivere controlli JavaScript basilari per campi obbligatori. Tutti i tipi di input (tranne button, image, reset, etc.) supportano `required`.
- **Attributo pattern**: consente di specificare una **espressione regolare** che il valore dell'input deve rispettare per essere considerato valido. Ad esempio, `<input type="text" pattern="[0-9]{3}-[A-Za-z]{2}">` richiede un formato di 3 cifre seguite da 2 lettere. Va spesso usato insieme a `required` (altrimenti il pattern viene applicato solo se qualcosa è inserito). Un caso comune: `<input type="url" required pattern="https?://.+">` per accettare solo URL che iniziano con http:// o https:// [57]. Se il testo non combacia con la regex, il form non si invia e appare un messaggio di errore personalizzabile via attributo `title`.
- **Placeholder**: l'attributo `placeholder="testo..."` su input e textarea mostra un suggerimento grigio quando il campo è vuoto, indicando all'utente quale valore inserire [58]. Appena l'utente digita, il placeholder scompare. Ad esempio, `<input type="text" placeholder="Cerca...">`. È utile come indicazione breve, ma **non sostituisce** la label (poiché scompare al focus e non viene letto come etichetta dallo screen reader).
- **Autofocus**: aggiungendo `autofocus` a un elemento di input, quel campo riceverà automaticamente il focus quando la pagina viene caricata [59]. Si usa per posizionare il cursore nel primo campo di un form (es. campo di ricerca o di login) così che l'utente possa iniziare a digitare subito senza cliccare. Nota: solo un elemento per documento può avere autofocus; se ce ne sono più, il primo in ordine DOM vince.
- **Autocomplete**: l'attributo `autocomplete` (valori "on"/"off" oppure valori specifici) indica se il browser può **autocompletare** il campo con dati memorizzati [60]. Ad esempio, per un campo nome utente `<input autocomplete="username">` aiuta il browser a suggerire lo username salvato, `autocomplete="new-password"` su un campo password può impedire riempimenti indesiderati. A livello di `<form>`, `autocomplete="off"` può disabilitare l'autofill per tutti i campi (utile ad es. per form di inserimento carte di credito o info sensibili che non si vogliono memorizzate).
- **Multipli valori**: l'attributo booleano `multiple` consente di selezionare/inserire più valori in alcuni input. Su `<input type="email" multiple>`, l'utente può inserire *più indirizzi email* separati da virgole (il browser valida ciascun indirizzo). Su `<input type="file" multiple>`, l'utente può scegliere più file da caricare in una volta sola. Questo attributo è molto utile per campi email (newsletter con più destinatari) e upload di file.
- **Elemento** `<datalist>`: HTML5 offre un modo nativo per fornire *suggerimenti di completamento* per un input text. `<datalist>` definisce una lista di opzioni predefinite, e l'`<input>` associato (con attributo `list="idDatalist"`) mostra il menu a tendina dei suggerimenti mentre si digita [61]. A differenza di `<select>`, l'utente non è obbligato a scegliere uno dei valori: può anche inserirne uno nuovo. Esempio: un campo `<input`

`list="citta">` con un `<datalist id="citta"><option value="Roma"><option value="Milano">...`. Man mano che l'utente scrive, vede le opzioni corrispondenti filtrate.

- **Elemento** `<output>`: serve a esporre il risultato di un calcolo o di un'azione effettuata dall'utente, tipicamente all'interno di un form (ma non necessariamente). Ad esempio, in un form di preventivo potresti sommare due campi e mostrare il totale in `<output name="totale">`. È un elemento *di sola visualizzazione* (l'utente non lo modifica direttamente). Può essere collegato a uno o più input tramite attributo `for="id1 id2"` (non obbligatorio) e avere un attributo `name` per essere inviato (invierebbe il valore calcolato) [62]. In pratica è uno span/div con semantica specifica di "risultato calcolato" [62].
- **Barra di progresso e misuratore**: HTML5 introduce `<progress>` e `<meter>` per rappresentare graficamente valori quantitativi. `<progress>` mostra l'avanzamento di un task in corso (es. progresso di upload, percentuale di completamento) ed è rappresentato come una barra di progresso dal browser [63]. Ha attributi `value` (quantità completata) e `max` (totale) per determinare la percentuale, e se `value` manca la barra viene mostrata in uno stato indeterminato (animazione continua). `<meter>` invece rappresenta una misura entro un intervallo noto (esempio classico: livello di batteria, punteggio o voto, intensità di qualcosa). Ha attributi `min`, `max`, `value` e opzionalmente `low`, `high`, `optimum` per indicare zone di valore (es. ottimo, accettabile, critico). Non va usato per i progressi di task (per quelli c'è progress), ma per valori statici o indicativi. Entrambi questi elementi vengono resi con stili default dal browser (che puoi personalizzare via CSS) e migliorano la semantica del dato rispetto a usare immagini o div.
- **Validazione CSS**: HTML5 non solo valida i form in automatico, ma espone anche pseudoclassi CSS per stilizzare gli input in base al loro stato di validità. Ad esempio `input:valid` e `input:invalid` permettono di applicare stili ai campi a seconda che abbiano un valore accettabile (rispetto a type, pattern, required, ecc.) [64]. Ci sono anche `:required` e `:optional` per selezionare campi obbligatori o facoltativi. Queste pseudoclassi consentono ad esempio di evidenziare in rosso i campi non validi o aggiungere un'icona di check vicino a quelli validati con successo, il tutto via CSS, senza JavaScript.
- **Attributi novità per moduli**: HTML5 ha introdotto anche altri attributi utili sui form. `autocomplete` l'abbiamo visto. C'è `novalidate` sul tag `<form>` per disabilitare la validazione HTML5 predefinita (magari perché vuoi gestirla via JavaScript personalizzato). Sugli `<input type="submit">` c'è `formnovalidate` per bypassare la validazione solo su quel submit (utile per un tasto "Salva bozza" che non richiede tutti i campi validi). L'attributo `form` su input/textarea/select permette di associare un campo a un form diverso dal suo padre (specificando l'id del form): in pratica campi fuori dal `<form>` ma che ne fanno parte. Infine, gli input file possono usare `accept="mime/type"` per indicare quali tipi di file accettare (es. `accept="image/png, image/jpeg"`). Sfruttare questi attributi migliora la UX e riduce la necessità di script custom per comportamenti standard.

Buone pratiche e performance

- **Usa markup semantico e pulito**: una buona pratica fondamentale è evitare l'uso improprio di tag e l'eccesso di wrapper. Ad esempio, non usare tabelle HTML per impaginare il layout (era comune nei anni '90, ma oggi layout e griglie si fanno con CSS). Le tabelle devono contenere solo dati tabulari: usare `<table>` per posizionare elementi compromette accessibilità (screen reader leggeranno celle non correlate) e responsività. Mantieni il codice snello: annidamenti eccessivi di `<div>` inutili o elementi vuoti rallentano il rendering e confondono la manutenzione. Un codice HTML5 **valido e semantico** è più facile da capire per tutti: sviluppatori, browser e motori di ricerca.

- **Evitare tag e attributi deprecati**: HTML5 ha eliminato molti elementi presentazionali (come `<font>`, `<center>`, `<big>`, `<blink>`, ecc.). Non usarli – oltre a non essere più standard, presentano contenuto in modo non accessibile. Usa i CSS per tutto ciò che è aspetto visivo (colori, font, allineamenti) e riserva l'HTML per la struttura e il significato. Ad esempio, invece di `<font color="red">Testo</font>` (deprecato), usa `<span class="rosso">Testo</span>` con CSS `.rosso { color: red; }`. Queste pratiche separano contenuto da presentazione, facilitando aggiornamenti e adattamenti multi-dispositivo.
- **Ordine dei contenuti e SEO/Accessibilità**: metti il contenuto importante in alto nel codice se possibile. Anche se con CSS puoi visualizzare elementi altrove, gli screen reader e i crawler leggono l'HTML nell'ordine. Avere la parte più rilevante (es. l'articolo, o i prodotti) prima nel DOM, e magari spostare nav e footer dopo, può migliorare leggermente la comprensione e assicurare che chi naviga da tastiera arrivi subito al nocciolo. In HTML5, grazie ai tag semantici, i motori riescono comunque a discernere sezioni, ma l'ordine logico rimane preferibile.
- **Caricamento CSS e JS in modo non bloccante**: i file CSS dovrebbero stare all'interno del `<head>` sotto forma di `<link rel="stylesheet">` così vengono caricati in parallelo al resto [65]. Non inserire troppe regole CSS inline nell'HTML (oltre a sporcare il markup, impediscono la cache e il riuso). Per gli script JavaScript esterni, mettili prima di `</body>` oppure usa l'attributo `defer` (o `async`) su `<script>` in `<head>`, in modo che **non blocchino il parsing della pagina** [66]. `defer` fa scaricare lo script in background e lo esegue *dopo* che l'HTML è stato caricato, mantenendo l'ordine dei vari script defer. `async` scarica in background ma poi esegue subito appena pronto (senza garantire ordine), quindi usalo per script indipendenti. Questo accelera il rendering perché l'HTML non si ferma in attesa dei file JS.
- **Ottimizzazione delle risorse**: riduci il peso di immagini e media – ciò spesso inizia fuori dall'HTML (ridimensionare e comprimere immagini, usare formati moderni come WebP/AVIF). Ma anche l'HTML offre strumenti: l'attributo `loading="lazy"` su `<img>` e `<iframe>` defersce il caricamento di immagini fuori dal viewport finché l'utente scrolla verso di esse [67]. Questo può migliorare nettamente il tempo di caricamento iniziale e risparmiare banda su immagini magari mai viste dall'utente. Tienilo presente per immagini grandi o gallerie sotto la piega. Inoltre, usa `srcset` / `sizes` per non caricare immagini più grandi del necessario su schermi piccoli.
- **Minimizza le richieste HTTP**: ogni risorsa esterna linkata (CSS, JS, immagini, font) comporta una richiesta. Troppi file possono rallentare la pagina. Strategie di build come concatenazione e minificazione dei CSS/JS riducono le richieste e il peso. Dal lato HTML, assicurati di non includere script o fogli di stile inutili in quella pagina. Usa attributi come `rel="preload"` su `<link>` per risorse fondamentali (es. un font o un hero image) così da caricarle in anticipo. Anche l'ordine delle inclusioni conta: metti i CSS prima di JS, così i CSS caricati non bloccheranno il **rendering critico** quando arrivano ritardati dai JS.
- **Cache e versioning**: sfrutta la cache del browser fornendo risorse statiche con intestazioni adeguate (questo si configura lato server). In HTML, un trucco comune è usare versioning nei file include, es: `<script src="app.js?v=1.2">`. Così, quando aggiorni app.js, cambi la versione nell'HTML e i browser scaricheranno la nuova, altrimenti useranno la cache. Ciò mantiene il sito veloce per utenti di ritorno.
- **Attributi HTML5 per performance**: alcuni attributi globali possono migliorare prestazioni/percezione. Ad esempio `decoding="async"` su `<img>` fa decodificare immagini in background, `fetchpriority="high"` su un `<img>` importante (LCP) può suggerire priorità alta di caricamento. Il tag `<link rel="prefetch">` può caricare in anticipo pagine collegate (es. l'articolo successivo) durante il tempo inattivo. Usali con giudizio per evitare spreco di banda o saturazione.
- **Separazione dei concern**: mantieni l'HTML focalizzato sul contenuto/struttura, il CSS sullo stile, e il JS sulla logica. Evita di mescolare: ad esempio, non usare attributi presentazionali (ormai rimossi, come bgcolor o align), non inserire grosso CSS nell'attributo `style` di elementi HTML

(meglio una classe), e non abusare di attributi on* (evento) inline per il JS. Invece di `<button onclick="addToCart()">` preferisci assegnare l'evento via script esterno, o almeno attraverso `addEventListener` in un blocco `<script>` separato. Ciò facilita manutenzione e debug.

- **Content Security Policy e sicurezza**: anche la sicurezza è best practice. Evita di inserire direttamente in HTML contenuti provenienti dall'utente senza sanitizzazione (rischio XSS). HTML5 introduce alcuni attributi utili: ad esempio `sandbox` sugli `<iframe>` per limitarne le azioni (disabilita script, modali, ecc., a meno che non si specifichi eccezioni in `allow`). Attributi come `rel="noopener"` su link target _blank prevengono accessi indesiderati alla `window.opener`. Considera l'aggiunta di intestazioni HTTP come CSP (Content Security Policy) che influiscono su cosa l'HTML può caricare/eseguire. Un codice HTML robusto tiene conto anche di questi aspetti.
- **Validazione e testing cross-browser**: anche se i browser moderni sono molto allineati, verifica il tuo HTML su vari browser e dispositivi. HTML5 ha un parsing definito per errori comuni, ma non abusarne: un markup errato potrebbe funzionare in un browser e rompersi in un altro. Usa strumenti di validazione per correggere tag non chiusi, attributi sconosciuti o annidamenti scorretti. Testa la pagina con JavaScript disabilitato per vedere se il contenuto essenziale è accessibile (progressive enhancement). Le best practice HTML puntano a un'esperienza di base funzionante senza scripting, arricchita poi da CSS e JS – questo rende il sito più resiliente e spesso migliora anche SEO e accessibilità.
- **Core Web Vitals**: benché siano metriche (Largest Contentful Paint, First Input Delay, Cumulative Layout Shift) più correlate a CSS/JS e infrastruttura, l'HTML ben strutturato incide. Ad esempio, dichiarare dimensioni fisse (width/height) per immagini e video nel markup previene layout shifts improvvisi (CLS) [21]. Posizionare in alto nel codice l'elemento più grande (tipicamente un'immagine o titolo) può aiutare il LCP. Riducendo la quantità di HTML inutilizzato, si può velocizzare il FID. Insomma, l'HTML è la base: un codice pulito e ottimizzato favorisce migliori performance percepite e misurate.

Integrazione con CSS e JavaScript (HTML e risorse esterne)

- **Collegare fogli di stile CSS**: per applicare stili, la pratica ottimale è includere file CSS esterni usando `<link rel="stylesheet" href="stili.css">` all'interno di `<head>`. Questo permette ai browser di scaricare il CSS in parallelo e cachearlo per altre pagine [65]. Non è necessario specificare `type="text/css"` (implicito in HTML5) [68]. Evita di usare l'elemento `<style>` nell'HTML per grandi quantità di CSS, salvo piccole porzioni critiche (al limite per above-the-fold inline CSS in ottica performance, ma va bilanciato). La separazione HTML/CSS mantiene il codice ordinato e riutilizzabile.
- **Inserire JavaScript**: l'HTML offre il tag `<script>` per includere codice JS. Può essere *esterno* (`<script src="script.js">`) o *inline* (codice diretto dentro). Per performance e mantenibilità, è meglio mettere la maggior parte degli script in file esterni (minificati e unificati). Inserisci i `<script>` appena prima della chiusura `</body>` così l'HTML viene caricato prima di eseguire gli script. In alternativa, in HTML5 puoi aggiungere l'attributo `defer` al tag script in `<head>` per ottenere lo stesso effetto (caricamento non bloccante, esecuzione dopo parsing HTML) [66]. `async` può essere usato per script indipendenti che possono eseguire appena pronti senza attendere l'ordine.
- **Modulo JavaScript**: HTML5 consente di specificare `type="module"` sul tag script (`<script type="module" src="main.js">`). Ciò indica che lo script è un modulo ES6, permettendo di usare `import` / `export`. I moduli sono eseguiti in modalità differita automaticamente (simile a defer) e ogni file ha il proprio scope (niente variabili globali a meno che non le definisci su window). Questo aiuta ad organizzare JS complesso in più file e caricare moduli secondo

necessità (anche dinamicamente via import). Ricorda che per i moduli, il percorso è relativo al file HTML o va usato un URL assoluto, e che moduli JS richiedono esecuzione da server (non file://).

- **Attributi globali per interazione**: HTML definisce alcuni attributi globali utili per integrazione. `id` e `class` sono fondamentali: assegna id univoci agli elementi se devi selezionarli con JavaScript o per deep linking (es. ancore) e usa class per gruppi di elementi a cui applicare stessi stili o comportamenti. Ad esempio, `<div id="gallery" class="box highlight">` può essere selezionato con CSS (`.box.highlight`) o JS (`document.getElementById('gallery')`). Mantieni nomi significativi per una manutenzione più semplice.
- **Attributi data-* (custom data)**: HTML5 permette di aggiungere attributi personalizzati che iniziano con `data-` su qualsiasi elemento, per memorizzare dati legati all'interfaccia [69] . Ad esempio `<li data-product-id="12345">Nome prodotto</li>` . Tramite JavaScript, puoi leggere questi valori con `element.dataset.productId` (il dataset espone ogni data-attribute in camelCase). Questo è molto utile per passare informazioni dal server/HTML al JS senza doverle scrivere nel codice JS statico. Usa data-* invece di usare attributi non standard o sovraccaricare classi con significato non visivo.
- **Eventi e comportamenti**: l'HTML permette di associare comportamenti JavaScript mediante attributi on *(es.* `onclick="func()"` , `onchange="..."` *), ma è meglio evitarli** per separare struttura e script. È preferibile usare JavaScript per agganciare event listener: ad esempio `document.querySelector('.btn').addEventListener('click', func)` . Tuttavia, sapendo che esistono: attributi come `onclick` , `oninput` , `onsubmit` vengono eseguiti quando l'evento corrispondente si verifica. Se li usi inline, assicurati di non violare concetti di sicurezza (es. evitare stringhe costruite dall'utente in onclick per prevenire XSS).
- **Includere risorse esterne terze parti**: spesso si incorporano librerie CSS/JS di terze parti via CDN. In HTML, ciò avviene con `<link>` o `<script>` puntati a URL esterni. Assicurati sempre di usare URL HTTPS e, se possibile, integrità SRI (`integrity="sha256-..." crossorigin="anonymous"`) per garantire che il file non sia alterato. Considera l'impatto di queste risorse su performance e privacy; ad esempio, le mappe di Google con `<script src="https://maps.googleapis.com/maps/api/js?...">` offrono funzionalità ma appesantiscono la pagina. Valuta di caricarle in modo condizionale o asincrono (`async defer`).
- `<base>` **tag**: se nella tua pagina molti link sono relativi a una certa base URL, puoi usare `<base href="https://esempio.com/cartella/">` nel head. Questo definisce la URL base per tutti i link e riferimenti relativi nella pagina. Ad esempio `<a href="pagina.html">` verrà interpretato come `https://esempio.com/cartella/pagina.html` . Può far comodo per evitare di scrivere percorsi assoluti lunghi. Puoi anche specificare `<base target="_blank">` per fare in modo che tutti i link si aprano in un'altra scheda per default. Nota: può impattare tutti i link/script/css, quindi usalo consapevolmente.
- **Integrazione social e embed**: HTML5 facilita l'inclusione di contenuti esterni. Per i video YouTube, mappe Google, post Facebook, ecc., spesso basta incollare uno snippet `<iframe>` fornito dal servizio. Mantieni però responsività usando attributi e CSS adatti (ad es. wrapper con `position: relative; padding-bottom: 56.25%; height: 0;` e iframe con `position: absolute; width:100%; height:100%;` per video 16:9). Per Facebook/Twitter, l'HTML5 può caricare script SDK che trasformano blocchi `<div>` in widget; cerca di farlo in modo asincrono per non bloccare il caricamento principale.
- **Contenuti fallback**: HTML5 consente di specificare contenuto alternativo dentro certi tag nel caso in cui la funzionalità non sia supportata. Ad esempio, tra `<video>...</video>` puoi mettere un link "Scarica il video" per browser che non supportano `<video>` [31] . Dentro `<canvas>...</canvas>` puoi inserire un messaggio tipo "Aggiorna il browser" o un grafico statico. Usa questa possibilità per migliorare la resilienza dell'applicazione: un HTML robusto fornisce sempre qualcosa all'utente.

- **Attributi globali utili**: alcuni attributi globali HTML5 spesso integrano CSS/JS. `hidden` (già menzionato) nasconde elementi fino a che magari via JS rimuovi l'attributo. `contenteditable="true"` fa sì che un elemento (p, div, ecc.) diventi editabile direttamente dall'utente (puoi poi salvare il risultato via script). `draggable="true"` permette di abilitare il drag&drop nativo HTML5 su un elemento (da gestire poi con eventi Drag & Drop API). `spellcheck="false"` può disattivare il controllo ortografico su campi dove non serve. Tali attributi arricchiscono l'interattività senza dover scrivere tutto da zero in script.
- **Compatibilità e polyfill**: quando integri nuove API o tag HTML5, verifica la compatibilità sui browser target. Ad esempio, `<dialog>` (per modali native) potrebbe non essere supportato su qualche browser più vecchio – in tal caso prevedi un polyfill (script che emula la funzionalità). Fortunatamente, la maggior parte dei tag e input HTML5 degradano in maniera silenziosa (ad esempio, un browser che non supporta `type="date"` tratterà quell'input come type="text"). È comunque buona norma testare funzionalità come Drag&Drop, ContentEditable, etc., e fornire alternative se necessario (ad es. bottone di fallback "Apri in popup" se `<dialog>` non funziona).
- **Inserimento di commenti e metadati**: nei file HTML, i commenti non vengono resi ma sono visibili nel sorgente pubblico, quindi evita di inserirvi informazioni sensibili (API key, credenziali, note riservate). Usa commenti per spiegare il codice a sviluppatori futuri, ma ricorda che fanno parte del peso della pagina scaricata (anche se minimo). Puoi usare commenti condizionali (funzionavano in IE per targettare versioni specifiche, es. `<!--[if IE 9]>` ... non più necessario oggi se non per rarissime eccezioni). Inoltre, puoi includere metadati addizionali come `<meta name="theme-color" content="#4285f4">` per impostare colore della barra mobile Chrome, oppure manifest PWA via `<link rel="manifest" href="manifest.json">`. Questi elementi migliorano l'integrazione della tua pagina con i dispositivi e piattaforme.

In sintesi, **HTML5** fornisce molti strumenti per creare pagine semantiche, accessibili, SEO-friendly e performanti. Applicare queste buone pratiche significa sfruttare appieno le potenzialità del linguaggio, facilitando il lavoro di browser e motori di ricerca e offrendo al contempo una migliore esperienza agli utenti finali. [70] [71]

---

[1] [2] [3] [4] [5] [6] [65] [66] [68] Document structure | web.dev
https://web.dev/learn/html/document-structure

[7] You are not required to close your

,

- , , or
  tags in ...
  https://blog.novalistic.com/archives/2017/08/optional-end-tags-in-html/

[8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] HTML5 e i tag semantici - ExTelos S.R.L.
https://extelos.com/blog/html5-e-i-tag-semantici/

[21] [24] [25] [27] HTML Video
https://www.w3schools.com/html/html5_video.asp

[22] [23] 4.8.6 The video element — HTML5
https://www.w3.org/TR/2011/WD-html5-20110113/video.html

[26] [29] [30] [31] HTML video and audio - Learn web development | MDN
https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Structuring_content/HTML_video_and_audio

28 HTML Canvas
https://www.w3schools.com/html/html5_canvas.asp

32 37 Accessibilità Web: linee guida e best practice per il 2025
https://www.uxinpills.it/accessibilita-web-linee-guida-e-best-practice-per-il-2025/

33 34 35 36 ARIA e HTML | web.dev
https://web.dev/learn/accessibility/aria-html?hl=it

38 39 40 41 42 43 44 45 46 47 48 49 50 70 71 14 Most Important Meta And HTML Tags You Need To Know For SEO
https://www.searchenginejournal.com/important-tags-seo/156440/

51 67 Browser-level image lazy loading for the web | Articles | web.dev
https://web.dev/articles/browser-level-image-lazy-loading

52 53 54 55 56 57 64 Form input con HTML 5: tutorial per la validazione | HTML.it
https://www.html.it/articoli/form-con-html5-i-nuovi-elementi-input/

58 HTML placeholder Attribute
https://www.w3schools.com/tags/att_placeholder.asp

59 HTML input autofocus Attribute - W3Schools
https://www.w3schools.com/tags/att_input_autofocus.asp

60 HTML autocomplete Attribute
https://www.w3schools.com/tags/att_autocomplete.asp

61 HTML datalist Tag
https://www.w3schools.com/tags/tag_datalist.asp

62 HTML output Tag
https://www.w3schools.com/tags/tag_output.asp

63 : The Progress Indicator element - HTML | MDN
https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Elements/progress

69 HTML Global data-* Attributes
https://www.w3schools.com/tags/att_global_data.asp