BONDVX · TEORIA DEL WEB

La teoria dietro i siti web

Fondamenta, rendering, framework, deploy — ancorate a uno stack reale

Questo documento è la **teoria dietro la costruzione dei siti**: il “come funziona davvero” e il “perché”, non il “cosa lanciare” (quello è nel manuale operativo). Presuppone che tu sappia programmare (variabili, funzioni, logica): salta le basi generiche e si concentra su ciò che è **specifico del web** — il browser, il rendering, i framework, il deploy, la performance, il 3D.

Come è ancorata a uno stack reale

Ogni capitolo chiude con un riquadro **« Nello stack BondVX** che collega la teoria a ciò che usiamo davvero nei siti che costruiamo: **Next.js 16 · React 19 · Tailwind v4 · React-Three-Fiber · Cloudflare Pages**. Così i concetti non restano astratti.

La legenda dei riquadri

Nota/analogia (teal) · **Da ricordare** (verde) · **Errore comune** (ambra) · **Approfondimento** (viola) · **Trappola/misconcezione** (rosso). Uso **analogie da analisi dati e segnali** dove aiutano a capire.

Indice

#	Contenuto
16	Grafica & 3D nel browser (WebGL)
17	Privacy & GDPR — la teoria
—	Ciclo di vita di un progetto + Glossario

PARTE I

Le fondamenta del web

Come il web muove le informazioni, i tre linguaggi di una pagina, e come il browser la trasforma in pixel.

Capitolo 1 — Come funziona il web

Il web è un enorme sistema **client-server**. Il **client** (il browser) chiede; il **server** risponde. Fra i due c'è un protocollo, **HTTP** (in versione cifrata, **HTTPS**): un ciclo di **richiesta** → **risposta**. È lo stesso schema di un'interrogazione a un database, o di una chiamata a uno strumento di misura che ti restituisce un pacchetto di dati.

1.1 · Cosa succede quando apri un indirizzo

1. **URL** — l'indirizzo (`https://acme.it/chi-siamo`): protocollo + dominio + percorso.
2. **DNS** — una rubrica globale traduce il dominio (`acme.it`) in un indirizzo **IP** numerico (dettagli nel cap. 11).
3. **Richiesta HTTPS** — il browser apre una connessione cifrata e chiede quella risorsa (verbo `GET`).
4. **Risposta** — il server manda uno **status code** (200 ok, 404 non trovato, 301 spostato…) + gli header + il corpo (di solito HTML).
5. **Rendering** — il browser interpreta l'HTML e costruisce la pagina (cap. 3).

> **HTTP è senza stato (stateless)**
> Ogni richiesta è indipendente: il server di suo non "ricorda" le precedenti. La memoria (login, carrello) si aggiunge sopra con **cookie**, token o storage nel browser. Tienilo a mente: è il motivo per cui l'autenticazione e le sessioni sono un layer *separato*, non una proprietà del protocollo.

> **» Nello stack BondVX**
> Quando apri un sito BondVX, non c'è un server applicativo che "gira": i file HTML sono già pronti e serviti da **Cloudflare Pages** (cap. 9 e 11). La richiesta la soddisfa il nodo CDN più vicino a te.

Capitolo 2 — I tre linguaggi e i loro ruoli

Una pagina web è fatta di tre linguaggi con tre mestieri distinti. Confonderli è la prima fonte di codice fragile. L'analogia: **HTML = lo scheletro**, **CSS = l'aspetto**, **JavaScript = il comportamento**.

Linguaggio	Mestiere	Esempio
HTML	Struttura e **significato** (semantica) del contenuto	`<h1>` titolo, `<nav>` menu, `<article>` articolo
CSS	Presentazione: colore, spazio, tipografia, layout	`color`, `display:grid`, `@media` (responsive)
JavaScript	Comportamento: interazione, logica, dati a runtime	click, `fetch` dati, aggiornare la pagina

Perché la SEMANTICA conta (non è estetica)

Usare il tag *giusto* (`<button>` per un bottone, `<h1>` per il titolo) non cambia come *appare* — cambia come la pagina viene **capita da chi non la vede con gli occhi**: i lettori di schermo (accessibilità, cap. 14) e i motori di ricerca (SEO, cap. 13). Un `<div>` stilizzato a bottone è invisibile a entrambi.

Errore comune: usare div per tutto

Il "div-itis" (ogni cosa è un `<div>`) produce pagine che sembrano giuste ma sono mute per accessibilità e SEO. La regola: **prima il tag semantico corretto**, poi lo stile.

» Nello stack BondVX

Nei siti BondVX il JSX di React è comunque HTML semantico sotto (`<header>`, `<main>`, `<section>`); Tailwind mette lo stile in classi utility sull'elemento, ma il **tag** resta la tua responsabilità semantica.

Capitolo 3 — Il DOM e il critical rendering path

Questo è **il modello mentale più importante** del capitolo. Il browser non “mostra l'HTML”: lo trasforma in strutture dati e le dipinge in una pipeline. Capirla spiega perché certe cose sono veloci e altre lente. È letteralmente una **pipeline di elaborazione**, come quelle che conosci dall'analisi segnali.

3.1 · La pipeline

1. **DOM** (Document Object Model) — il browser fa il parsing dell'HTML in un **albero** di nodi manipolabile dal codice.
2. **CSSOM** — idem per il CSS: un albero delle regole di stile.
3. **Render Tree** — DOM + CSSOM fusi: solo ciò che è visibile, con i suoi stili calcolati.
4. **Layout (reflow)** — il browser calcola *dove* e *quanto grande* è ogni box. È la fase costosa.
5. **Paint** — riempie i pixel (colori, testo, ombre) in livelli.
6. **Composite** — assembla i livelli sullo schermo (la GPU aiuta qui).

Reflow vs repaint — perché le animazioni scattano

Cambiare una proprietà che tocca la **geometria** (`width`, `top`, `margin`) forza un nuovo **Layout** (reflow) → costoso. Cambiare `transform` o `opacity` tocca solo il **Composite** → la GPU lo fa a 60fps. **Regola pratica:** anima `transform`/`opacity`, non `left`/`width`.

Il JS può bloccare il disegno

Il parser incontra uno `<script>` sincrono e **si ferma** finché non l'ha eseguito (potrebbe modificare il DOM). Per questo gli script vanno `defer`/`async` o in fondo: è alla base della teoria di performance (cap. 12).

» Nello stack BondVX

React costruisce e aggiorna il DOM per te (cap. 7), ma le regole della pipeline restano: le gotcha 3D del manuale operativo (bagliore che scatta, canvas che ridisegna) sono **reflow/paint** mascherati.

PARTE II

II

Dallo statico all'app

Il layout e il colore come sistemi, il JavaScript nel browser, e il salto al modello a componenti di React.

Capitolo 4 — CSS moderno & teoria del layout

Il CSS ha un modello preciso; sembra “magia” solo finché non lo vedi. Tre pilastri: il **box model**, il **flusso** (come gli elementi si dispongono di default) e i **sistemi di layout** (Flexbox, Grid) che lo governano.

4.1 · Box model & flusso

Ogni elemento è una scatola: **content** → **padding** → **border** → **margin** (dall'interno all'esterno). Di default gli elementi **block** vanno uno sotto l'altro, gli **inline** in linea. Con `box-sizing: border-box` la larghezza include padding e bordo (più intuitivo).

4.2 · Flexbox vs Grid

Sistema	A cosa serve	Mentalità
Flexbox	Disporre elementi lungo **UN** asse (riga o colonna)	1D: distribuzione + allineamento
Grid	Layout su **DUE** assi (righe e colonne) insieme	2D: griglia, aree nominate

4.3 · Cascade, specificity, unità

- **Cascade** — se due regole toccano la stessa proprietà, vince la più specifica (poi l'ultima scritta).
- **Specificity** — inline > `#id` > `.classe` > tag. `!important` scavalca (usarlo è un campanello d'allarme).
- **Unità** — `rem` (relativa al font-size radice) per scalabilità, `%`/`vw`/`vh` per il viewport, `px` per valori fissi.
- **Custom properties** — `--brand: #0891b2` → variabili CSS riusabili: sono la base dei **design token**.

> **» Nello stack BondVX**
> **Tailwind v4** è *utility-first*: invece di scrivere CSS, componi classi (`flex gap-4 rounded-xl`) che **mappano 1:1** su queste proprietà. Non è un linguaggio nuovo — è CSS con nomi brevi. Nei nostri progetti `brand.ts`/`globals.css` definiscono i token (le custom properties) che le utility usano.

Capitolo 5 — Colore & tipografia

Colore e testo non sono “gusto”: hanno una matematica. Sapere come è fatto un colore e come scala il testo ti fa scegliere valori che **funzionano** invece di tirare a indovinare.

5.1 · Spazi colore: sRGB vs OKLCH

sRGB (`#0891b2`, `rgb()`) descrive il colore come miscela di rosso/verde/blu: ottimo per lo schermo, pessimo per *ragionarci* (schiarire del 10% non dà un 10% percepito). **OKLCH** lo descrive come **L**uminosità · **C**roma · **H**ue (tonalità) in modo **percettivamente uniforme**: cambiare L di un tot cambia la luminosità *vista* di quel tot. È come passare dal dominio del tempo a quello della frequenza: stessi dati, ma su assi che hanno senso per l'operazione che devi fare.

> **Perché nei siti si usa OKLCH per i token**
> Con OKLCH generi scale di colore coerenti (stessa tonalità, luminosità a gradini) e prevedi il **contrasto** — che è un requisito di accessibilità, non un'opzione (cap. 14).

5.2 · Contrasto WCAG & tipografia

- **Contrasto WCAG** — il rapporto di luminanza testo/sfondo deve essere ≥ **4.5:1** (testo normale) o ≥ 3:1 (testo grande). Sotto, diventa illeggibile per molti.
- **Scala tipografica** — le dimensioni del testo seguono una progressione (es. ×1.25 per livello), non valori a caso.
- `clamp(min, ideale, max)` — fa scalare il testo col viewport restando entro limiti: tipografia fluida senza decine di media query.
- **Font loading** — i font vanno caricati bene o causano “salti” (CLS, cap. 12); meglio a **build-time**, non da CDN a runtime.

> **Trappola: oklch() passato a una libreria 3D**
> `oklch()` è per il **CSS**. Se lo passi ai material di three.js/R3F, `THREE.Color` non lo capisce e diventa **bianco**. Ai material serve HEX/sRGB (è la gotcha del cap. 16 e del manuale operativo).

> **» Nello stack BondVX**
> I token colore nei siti sono in OKLCH dentro `globals.css`; i font sono self-hostati via `next/font` (build-time → niente salti, niente richiesta a `fonts.googleapis.com`, GDPR-clean, cap. 17).

Capitolo 6 — JavaScript nel browser

Il JS che gira nel browser ha una caratteristica che spiega quasi tutto il suo comportamento: è **single-thread**. Un solo “esecutore” alla volta. Come un DAQ con un solo canale: per non bloccarsi mentre aspetta, usa una coda.

6.1 · L'event loop

Il motore ha uno **stack** (ciò che sta eseguendo ora) e delle **code** (cose da fare dopo). Le operazioni lente (un `fetch`, un timer, un click) non bloccano: vengono affidate all'ambiente, e la loro **callback** entra in coda. Quando lo stack è vuoto, l'**event loop** pesca dalla coda. Le **microtask** (le Promise) hanno priorità sulle **macrotask** (timer, eventi).

> **Async non vuol dire parallelo**
> `async/await` e le Promise NON creano thread paralleli: **riordinano** il lavoro su un unico thread, così l'attesa (rete, disco) non congela l'interfaccia. Se fai un calcolo pesante *sincrono*, l'UI si blocca lo stesso — per quello servono i Web Worker (thread separati veri).

6.2 · Dal DOM manipolato a mano ai framework

Il JS “classico” seleziona nodi (`document.querySelector`) e li modifica a mano. Su una pagina piccola va bene; su un'app con molti stati diventa un incubo di sincronizzazione (“quale pezzo di DOM devo aggiornare quando cambia questo dato?”). È il problema che i framework risolvono (cap. 7).

> **» Nello stack BondVX**
> Nei siti scrivi pochissimo DOM a mano: React tiene il DOM in sync col tuo **stato**. Il JS “vanilla” resta per micro-interazioni (es. i listener `mousemove` del tilt 3D del cap. 16).

Capitolo 7 — Il modello a componenti & React

React è il salto da **imperativo** (“prendi quel nodo e cambialo”) a **dichiarativo** (“data questa situazione, la UI è FATTA COSÌ”). Tu descrivi il risultato; React trova le modifiche minime da applicare. È un cambio di funzione: da *procedura* a *funzione pura* stato → vista.

7.1 · Componenti, props, state

- **Componente** — una funzione che ritorna un pezzo di UI. Si compongono come mattoncini (`<Header/>` dentro `<Page/>`).
- **Props** — i dati che un componente **riceve** dall'alto (in sola lettura). Come i parametri di una funzione.
- **State** — i dati che un componente **possiede** e che, cambiando, ne provocano il re-render.

7.2 · Virtual DOM & reconciliation

Quando lo stato cambia, React ricalcola una rappresentazione in memoria della UI (il **Virtual DOM**), la **confronta** con la precedente (**reconciliation**) e applica al DOM reale **solo il diff**. Così eviti di riscrivere tutta la pagina e tocchi il DOM (costoso) il meno possibile.

Da ricordare: ri-render ≠ ridisegno del browser

“React ri-renderizza” significa “ricalcola il Virtual DOM ed eventualmente aggiorna qualche nodo” — non “ridipinge tutto lo schermo”. Sono due livelli diversi (React sopra, la pipeline del cap. 3 sotto).

Approfondimento: gli hooks in una riga

Gli **hook** (`useState`, `useEffect`, …) sono funzioni che danno “memoria” e “effetti collaterali” a un componente-funzione: `useState` = uno stato, `useEffect` = “esegui questo dopo il render / quando cambia X”.

» Nello stack BondVX

I siti usano **React 19**. Il template `brand.ts` è la “fonte di verità” passata come dati ai componenti; cambiare un valore lì ri-renderizza le sezioni che lo usano.

PARTE III

III

Architettura, framework, rendering

Cosa fa un framework, dove e quando nasce l'HTML (CSR/SSR/SSG/ISR), e da dove arrivano i dati.

Capitolo 8 — Framework & build tools

Il codice che scrivi **non** è il codice che arriva al browser. In mezzo c'è una **toolchain** che traduce, unisce e ottimizza. Capirla demistifica il “perché serve un build”.

8.1 · Cosa fa un build tool

- **Transpiling** — traduce ciò che il browser non capisce (JSX di React, TypeScript, CSS moderno) in JS/CSS standard.
- **Bundling** — unisce decine di file/moduli in pochi pacchetti che il browser scarica bene.
- **Tree-shaking** — elimina il codice non usato (come il dead-code elimination di un compilatore).
- **Minify + hashing** — comprime e mette un “hash” nel nome file (`app.9f2a.js`) per il caching (cap. 12).

8.2 · Cosa fa un framework (Next.js)

Una **libreria** (React) la chiami tu. Un **framework** (Next.js) chiama il tuo codice: ti dà la struttura (routing, rendering, build, ottimizzazioni) e tu riempi gli spazi. Next.js aggiunge a React il routing basato su file, le strategie di rendering (cap. 9), l'ottimizzazione di immagini/font e il build.

> **» Nello stack BondVX**
> Lo stack è **Next.js 16 + React 19**, con **Tailwind v4** che processa il CSS via **lightningcss** (un compilatore CSS velocissimo). Il comando di build produce la cartella statica che poi va in deploy.

Capitolo 9 — Le architetture di rendering

“Dove e quando si trasforma il codice in HTML?” La risposta definisce l'architettura. Quattro sigle coprono lo spettro; sono la decisione tecnica più importante di un progetto.

Sigla	Quando si genera l'HTML	Pro / contro
CSR	Nel **browser**, dopo aver scaricato il JS (client-side rendering)	Molto dinamico; ma prima schermata lenta, SEO fragile
SSR	Sul **server, a ogni richiesta**	Sempre fresco + buono per SEO; ma serve un server sempre acceso
SSG	Una volta, al **build** (static site generation)	Velocissimo, sicuro, cache-abile; ma i dati sono “fotografati” al build
ISR	SSG + **rigenerazione** periodica/on-demand	Il meglio dei due; ma richiede l'infrastruttura del framework

9.1 · Hydration

SSR e SSG mandano **HTML già pronto** (veloce da mostrare, ottimo per SEO). Ma è “statico”: per renderlo interattivo, il browser scarica il JS di React che si “attacca” all'HTML esistente, ricollega gli event handler e ricostruisce lo stato. Questo processo si chiama **hydration** (idratazione): prendi lo scheletro secco e lo “bagni” rendendolo vivo.

> **Da ricordare: statico non vuol dire “senza JavaScript”**
> Un sito SSG serve HTML pre-costruito **e** può essere pienamente interattivo dopo l'hydration. “Statico” si riferisce a *quando* si genera l'HTML (al build), non alla mancanza di dinamismo.

> **» Nello stack BondVX**
> I siti usano `output: 'export'` → **SSG puro**: tutte le pagine pre-generate al build, zero server a runtime. **Perché:** massima velocità (CDN edge), costo quasi nullo, superficie d'attacco minima, GDPR più semplice. **Prezzo da pagare:** niente SSR/ISR a runtime → i dati che cambiano spesso (es. blog) arrivano da un backend separato letto lato client (cap. 10), non dal render della pagina.

Capitolo 10 — Routing, dati e stato

10.1 · Routing

Il **routing** associa un URL a una vista. In Next.js (App Router) è basato sui **file**: la struttura delle cartelle in `src/app/` È la mappa degli URL (`app/chi-siamo/page.tsx` → `/chi-siamo`). Niente tabella di rotte da mantenere a mano.

10.2 · Server components vs client components

React 19/Next distinguono i componenti che girano **al build/sul server** (possono leggere dati, non hanno interattività) da quelli **client** (marcati `'use client'`, hanno stato/eventi). Regola pratica: **client solo dove serve interazione** (un form, un menu, il 3D); tutto il resto resta server/statico e non pesa sul bundle.

10.3 · Da dove vengono i dati

- **Build-time** — dati “fotografati” quando costruisci (contenuti fissi, liste stabili). Finiscono nell'HTML statico.
- **Runtime, lato client** — dati che cambiano (articoli, form) letti dal browser via `fetch` da un backend/API dopo il caricamento.
- **Stato locale** — dati effimeri della UI (menu aperto, tab attivo) che vivono e muoiono nel componente.

> **» Nello stack BondVX**
> Backend dei siti = **Supabase** (database + API). Con l'export statico, i contenuti dinamici (es. il blog automatico) si **scrivono** in Supabase e la pagina li **legge lato client**: così un nuovo articolo appare senza ricostruire il sito (cap. 9).

PARTE IV

IV

Consegnare e far vivere un sito

Hosting e rete, performance misurabile, SEO, accessibilità, sicurezza, grafica 3D e privacy.

Capitolo 11 — Hosting, DNS, HTTPS, CDN edge

“Mettere online” un sito è quattro cose distinte: dove vivono i file (**hosting**), come ti trovano (**DNS**), come la connessione è sicura (**HTTPS**), come è veloce ovunque (**CDN**).

11.1 · DNS — la rubrica di Internet

Record	Traduce	Esempio
A / AAAA	dominio → indirizzo IP (v4 / v6)	`acme.it` → `104.21.x.x`
CNAME	dominio → un altro dominio (alias)	`www` → `acme.pages.dev`
TXT	testo di verifica (proprietà, email)	verifica dominio, SPF/DKIM

Cambiare i record al **registrar** (dove hai comprato il dominio) è l'unico passo davvero manuale del go-live; la propagazione può richiedere minuti/ore.

11.2 · HTTPS/TLS & CDN edge

- **TLS/HTTPS** — il sito presenta un **certificato** firmato da un'autorità (CA): prova la sua identità e cifra la connessione. Senza, i browser marcano il sito “non sicuro”.
- **Static hosting** — servire file già pronti (HTML/CSS/JS/immagini) senza un server applicativo che “gira”: veloce e robusto.
- **CDN / edge** — copie dei file su tanti nodi (POP) sparsi nel mondo: l'utente scarica dal nodo **più vicino** → latenza bassa. È come avere il dato in cache vicino al sensore invece che a chilometri.

> **» Nello stack BondVX**
> **Cloudflare Pages** = static hosting + CDN edge globale + HTTPS automatico, in uno. Il tuo sito statico (cap. 9) è la combinazione ideale per questo modello: nessun server, servito dall'edge, certificato incluso.

Capitolo 12 — Performance & Core Web Vitals

La performance è misurabile, non “a sensazione”. Google ha standardizzato tre metriche — i **Core Web Vitals** — che pesano anche sul ranking (cap. 13). Sono, di fatto, tre indicatori di qualità del segnale “esperienza utente”.

Metrica	Misura	Buono se
LCP	**L**argest Contentful Paint — quanto ci mette il contenuto principale ad apparire (caricamento)	≤ 2.5 s
CLS	**C**umulative Layout Shift — quanto la pagina “salta” mentre carica (stabilità visiva)	≤ 0.1
INP	**I**nteraction to Next Paint — quanto è reattiva a click/tap (reattività)	≤ 200 ms

12.1 · Le leve teoriche

- **Critical path corto** — meno CSS/JS bloccante prima del primo disegno (cap. 3).
- **Lazy loading** — carica dopo ciò che non serve subito (immagini sotto la piega, il 3D pesante).
- **Caching + asset immutabili** — i file con hash nel nome (cap. 8) si cachano “per sempre”: la seconda visita è istantanea.
- **Immagini & font** — formati moderni, dimensioni giuste, font a build-time: sono le cause n.1 di LCP/CLS scarsi.
- **Code splitting** — spezza il bundle così ogni pagina scarica solo il suo codice.

> **Da ricordare: il candidato LCP**
> Su un sito con hero video/3D, l'elemento LCP dovrebbe essere un'**immagine poster** leggera e self-hostata, MAI il video o il canvas 3D (che sono pesanti). È il principio dietro il “budget di performance” dei siti premium.

> **» Nello stack BondVX**
> Next ottimizza bundle, immagini e code-splitting di default; il 3D (React-Three-Fiber) va caricato in **lazy** (`next/dynamic, ssr:false`) così non blocca il primo paint (cap. 16).

Capitolo 13 — SEO — come i motori vedono un sito

La SEO non è “trucchi”: è rendere il sito **comprensibile alle macchine**. Un motore fa tre cose in sequenza.

1. **Crawling** — un bot scopre e scarica le pagine (seguendo link e sitemap).
2. **Indexing** — le analizza e le archivia, capendo di cosa parlano (qui contano semantica e contenuto).
3. **Ranking** — le ordina per una query, pesando rilevanza, autorevolezza e **esperienza** (Core Web Vitals, cap. 12).

13.1 · Gli ingredienti tecnici

- **Meta tag** — `<title>` e `meta description`: ciò che appare nei risultati.
- **Sitemap & robots** — `sitemap.xml` (mappa delle pagine) e `robots.txt` (cosa il bot può visitare).
- **Structured data (JSON-LD)** — dati in formato macchina (`schema.org`) che spiegano “questa è un'azienda / un articolo / un evento” → abilita i rich result.
- **HTML semantico** (cap. 2) e, per i multilingua, `hreflang` (quale lingua per quale paese).

> **Perché lo statico aiuta la SEO**
> Con SSG (cap. 9) il bot riceve **HTML già completo** di contenuto: niente rischio che il testo “appaia solo dopo il JavaScript”. È uno dei motivi per cui i siti-vetrina sono statici.

> **» Nello stack BondVX**
> Nel flusso di build c'è uno step SEO dedicato: sitemap + feed + `robots.txt` + JSON-LD + IndexNow (ping ai motori). Le landing per-sede reale servono la SEO locale (con coordinate vere, mai inventate).

Capitolo 14 — Accessibilità (a11y)

L'accessibilità è rendere il sito usabile da **tutti**, inclusi chi usa lettori di schermo, solo tastiera, o ha problemi di vista/motori. Non è un extra: è teoria di base, e in più aiuta SEO e usabilità per tutti. Lo standard di riferimento è **WCAG**.

- **Semantica** (cap. 2) — il tag giusto costruisce l'**accessibility tree** che il lettore di schermo legge. È il 90% del lavoro.
- **Tastiera** — tutto ciò che si fa col mouse deve farsi con Tab/Invio; il **focus** deve essere visibile.
- **Contrasto** — ≥ 4.5:1 (cap. 5): testo grigio-chiaro su bianco è un classico fallimento.
- **ARIA** — attributi che aggiungono significato QUANDO il tag nativo non basta (es. un widget custom).
- `prefers-reduced-motion` — rispetta chi ha disattivato le animazioni (vertigini): niente motion forzato.

> **Trappola: “metto ARIA ovunque”**
> ARIA **sbagliato è peggio di niente ARIA**: confonde i lettori di schermo. La gerarchia corretta è **prima HTML semantico nativo** (`<button>`, `<nav>`), ARIA solo come ultima risorsa per i casi che il tag nativo non copre.

> **» Nello stack BondVX**
> Nei siti: componenti con tag semantici, `prefers-reduced-motion` obbligatorio su ogni pattern di motion/3D (cap. 16), e il gate di QA controlla anche il contrasto.

Capitolo 15 — Sicurezza web di base

Non devi essere un esperto di security, ma il **modello mentale delle minacce** ti evita gli errori da principiante. Tre concetti bastano per il 90% dei siti.

- **HTTPS ovunque** (cap. 11) — senza, i dati viaggiano in chiaro e la pagina è manomettibile in transito.
- **Same-origin policy & CORS** — di default il browser **vieta** a una pagina di leggere dati di un altro dominio; un server può **consentirlo** esplicitamente con gli header **CORS**. È un muro di sicurezza, non un bug.
- **Secrets solo lato server** — le chiavi API/segreti non vanno MAI nel codice che arriva al browser (è tutto ispezionabile): stanno in variabili d'ambiente sul server/worker.

15.1 · Le due minacce classiche

Attacco	Cos'è	Difesa
XSS	Iniettare script maligno in una pagina (via input non ripulito) che gira nel browser della vittima	**Sanitizzare** l'input, escape dell'output, **CSP**
CSRF	Ingannare il browser di un utente loggato per fargli fare una richiesta non voluta	Token anti-CSRF, cookie `SameSite`

Approfondimento: CSP

La **Content-Security-Policy** è un header che dice al browser “esegui script/stili SOLO da queste sorgenti”: anche se un XSS riesce a iniettare codice, la CSP può impedirne l'esecuzione. È una rete di sicurezza, non un sostituto della sanitizzazione.

» Nello stack BondVX

Un sito statico ha una **superficie d'attacco piccola** (nessun server applicativo, nessun DB esposto direttamente). I segreti (chiavi Supabase service-role, API key) vivono nei **Worker**/variabili d'ambiente, mai nel bundle; gli `artifact` e le dashboard sono CSP-safe.

Capitolo 16 — Grafica & 3D nel browser (WebGL)

Il browser sa disegnare ben oltre testo e box. Tre livelli di grafica, dal più semplice al più potente.

Tecnica	Modello	Quando
SVG	**Vettoriale**, retained: forme sono nodi nel DOM, scalano senza perdere qualità	Loghi, icone, illustrazioni nitide
Canvas 2D	**Raster**, immediate: dipingi pixel via codice, nessun nodo persistente	Giochi 2D, grafici, effetti particellari
WebGL	Accesso alla **GPU** per il 3D (e 2D pesante)	Scene 3D, shader, esperienze immersive

16.1 · La pipeline 3D (il modello mentale)

Una scena 3D è sempre gli stessi ingredienti — è come montare un set fotografico:

- **Scena** — il “mondo” che contiene tutto.
- **Geometry** — la forma: un insieme di vertici (un cubo, un casco importato come mesh GLB).
- **Material** — come la superficie reagisce alla luce (colore, metallico, rugosità); dietro ci sono gli **shader** (programmi che girano sulla GPU).
- **Mesh** — geometry + material insieme = l'oggetto visibile.
- **Camera** — il punto di vista (posizione, prospettiva).
- **Luci** — senza luce, un material fisico è nero.

16.2 · Animazione

Per il movimento: le **CSS transitions/keyframes** (dichiarative, ottime per UI; se animano `transform`/`opacity` restano sul composite — cap. 3) o `requestAnimationFrame` (una callback prima di ogni ridisegno, ~60fps) per il controllo fine e il 3D. Mai `setInterval` per animare: non è sincronizzato col refresh dello schermo.

Approfondimento: three.js e React-Three-Fiber

WebGL è a bassissimo livello. **three.js** lo incapsula in oggetti comodi (Scene, Mesh, Camera). **React-Three-Fiber (R3F)** ti fa descrivere la scena 3D come **componenti React** (`<mesh>`, `<Canvas>`) — stesso modello dichiarativo del cap. 7, applicato al 3D.

Trappola: l'HDRI da CDN e i colori oklch

Due misconcezioni costose: (1) `<Environment preset="…">` scarica un file di illuminazione da una CDN a runtime (lento, non GDPR-clean) → usa una luce procedurale; (2) i colori `oklch()` diventano **bianco** nei material (cap. 5). Entrambe sono nel manuale operativo.

» Nello stack BondVX

I siti premium usano **three 0.184 + R3F 9.6 + drei 10.7**, caricati in lazy e con fallback procedurale quando gli asset generati non ci sono. Il 3D non deve mai bloccare il primo paint (cap. 12).

Capitolo 17 — Privacy & GDPR — la teoria

Il GDPR non è “il banner dei cookie”: è una teoria del **trattamento dei dati personali**. Capirla ti dice *perché* il codice fa certe scelte.

- **Dato personale** — qualsiasi cosa identifichi una persona (nome, email, ma anche IP e ID dei cookie).
- **Base giuridica** — per trattare un dato serve un motivo legale; per marketing/tracciamento è il **consenso** (una delle sei basi).
- **Cookie & tracciamento** — analytics e pixel pubblicitari leggono/scrivono dati sul dispositivo → richiedono consenso **preventivo** (ePrivacy).
- **Consent Mode** — gli strumenti (es. Google) partono in stato “negato” e si attivano solo dopo l'OK dell'utente.

Il principio tecnico chiave

Un sito conforme **non spara richieste a terze parti PRIMA del consenso**. Da qui la regola di **self-hostare tutto** (font, video, asset 3D, illuminazione): se non chiami domini esterni, non c'è trasferimento di dati da consentire prima. La conformità nasce nell'**architettura**, non solo nel banner.

Distinzione: cookie/consenso vs identità legale

Sono due cose diverse. Il **consenso cookie** (banner + policy) riguarda i dati dei visitatori. L'**identità legale** (Art. 2250 c.c.: ragione sociale, sede, P.IVA in footer) riguarda chi **gestisce** il sito/azienda. Entrambe servono, ma rispondono a norme diverse.

» Nello stack BondVX

I siti nascono GDPR-ready: banner + policy + Consent Mode v2, analytics attivati solo post-consenso, e asset self-hostati by design. Un gate automatico blocca il go-live pagato se manca la parte cookie/policy.

Capitolo 18 — Chiusura: ciclo di vita + glossario

18.1 · Il ciclo di vita di un progetto web (in astratto)

Tutta la teoria dei capitoli precedenti si dispone lungo un percorso:

1. **Concept** — obiettivo, pubblico, tipo di sito (che architettura di rendering serve, cap. 9).
2. **Design system** — token di colore/tipografia/spazio PRIMA delle pagine (cap. 4-5).
3. **Build** — struttura semantica + componenti (cap. 2, 7, 8).
4. **Contenuti** — testo, immagini ottimizzate, dati (cap. 10, 12).
5. **GDPR & a11y & SEO** — conformità e comprensibilità (cap. 13, 14, 17).
6. **QA** — performance (Core Web Vitals), layout, contrasto (cap. 12).
7. **Deploy** — hosting + DNS + HTTPS + CDN (cap. 11).
8. **Vita** — aggiornamenti, contenuti nuovi, monitoraggio.

18.2 · Glossario

Termine	Significato in una riga
DOM	L'albero di nodi in cui il browser trasforma l'HTML, manipolabile dal codice.
CSSOM	L'equivalente del DOM per le regole CSS.
Critical rendering path	La pipeline DOM→CSSOM→render tree→layout→paint→composite.
Reflow / repaint	Ricalcolo di geometria (costoso) / ridisegno di pixel.
Cascade / specificity	Le regole che decidono quale stile CSS vince.
Virtual DOM	La copia in memoria della UI che React confronta per aggiornare il minimo.
Reconciliation	Il diff fra vecchio e nuovo Virtual DOM.
Event loop	Il meccanismo che fa girare il JS single-thread senza bloccarsi sulle attese.
Bundler / transpiler	Strumenti che uniscono i moduli / traducono JSX-TS-CSS moderno in standard.
Tree-shaking	Rimozione del codice non usato dal bundle.
CSR / SSR / SSG / ISR	Dove e quando si genera l'HTML (browser / server-per-richiesta / build / build+rigenera).
Hydration	Rendere interattivo l'HTML statico attaccandogli il JS di React.
Server / client component	Componente che gira al build-server (dati, no interazione) vs nel browser (stato, eventi).
DNS	La rubrica che traduce un dominio in indirizzo IP.
TLS / HTTPS	Cifratura + certificato che rende sicura e identificata la connessione.
CDN / edge / POP	Copie dei file su nodi vicini all'utente per servirli velocemente.
Core Web Vitals	LCP (caricamento), CLS (stabilità), INP (reattività).
JSON-LD / schema.org	Dati strutturati per far capire ai motori di cosa parla la pagina.

Termine	Significato in una riga
WCAG / ARIA	Standard di accessibilità / attributi che aggiungono significato quando il tag nativo non basta.
Same-origin / CORS	Il divieto di default di leggere dati cross-dominio / il modo per consentirlo.
XSS / CSRF	Iniezione di script maligno / richiesta forzata a nome di un utente loggato.
CSP	Header che limita le sorgenti di script/stili consentite.
WebGL / three.js / R3F	Accesso GPU per il 3D / la libreria che lo incapsula / il suo binding React.
Consent Mode	Gli strumenti partono “negato” e si attivano solo dopo il consenso.

Manutenzione: questo PDF si rigenera con `python3 scripts/build-guida-teoria-siti-pdf.py`. Il manuale **operativo** (comandi, prezzi, deploy) è il documento gemello `guida-creazione-siti.pdf`.