CSS and DOM

Assistive tech awareness: screen readers (JAWS/NVDA/VoiceOver), magnifiers, switch/eye
Basic: CSS selector forms: `tag`, `.class`, `#id`, `tag.class`, `[attr=value]`, `A B` descendant, `A > B` child, `A + B` adjacent sibling, `A ~ B` general sibling, `*` all elements.
Classes: `el.classList.add/remove/toggle`.
Common patterns: Common CSS selector combos: `.btn.primary`, `ul > li.active`, `a[href^="/"]`, `input:focus` (use for targeting state/structure precisely).
Contrast: meet WCAG contrast; don't rely on color alone (add text/icons/patterns).
Control flow: `event.preventDefault()`, `event.stopPropagation()`.
Events: pass a function reference to `addEventListener` (not `fn()`).
Focal point: strongest visual attractor guiding initial attention.
Forms: call `preventDefault()` on submit before async work.
Headings: logical order (h1→h2→h3).
Keyboard: everything reachable/actionable by keyboard; logical tab order; visible focus.
Landmarks/roles: `<main>`, `<nav>`, `<header>`, `<footer>`, `role="dialog"` with focus.
Pitfall: shallow copy keeps nested references; mutating nested objects mutates both copies.
Query: `document.querySelector(css)`, `querySelectorAll`, `getElementById`.
Specificity order (highest → lowest): CSS specificity: inline style > `#id` > `.class`/`[attr]`/`:pseudo-class` > `tag`/`::pseudo-element` > `*` (higher wins when rules conflict).
Text vs HTML: `.textContent` (safe) vs `.innerHTML` (parses HTML; XSS risk).
Visual scanning: F-pattern (text-heavy), Z-pattern (simple hero pages); use hierarchy/contrast.
WAVE: automated accessibility audit tool; finds common WCAG issues quickly.
axe: automated accessibility audit tool (browser extension/dev tooling).
WIMP: Windows, Icons, Menus, Pointer.
WCAG's POUR: Perceivable, Operable, Understandable, Robust; use A/AA/AAA framing.
Accessibility grouping: group related controls/labels visually and semantically so scanning and SR navigation are easier.
Alt text: describe function/meaning.

HTTP, Fetch, JSON, CORS

A fetch doesn't update UI unless you set state or update the DOM.: fetch changes data, not the UI; you must re-render/modify DOM yourself.
Configure method,h,b: `(url, { method, headers, body: JSON.stringify(x) })`.
Credentials: set `credentials: "include"` for cookie auth; server must allow via CORS.
CORS allow headers: server replies with `Access-Control-Allow-Origin/Methods/Headers` & `Allow-Credentials`.
CORS preflight: browser sends `OPTIONS` with `Access-Control-Request-*` for non-simple requests.
FormData: use `new FormData(formEl)`; browser sets multipart boundary; don't manually set `Content-Type`.
Headers (Request): `Accept`, `Authorization: Bearer <token>`, `Content-Type: application/json`, custom `X-CS571-ID`.
Headers (Response): `Content-Type`, `Cache-Control`, `ETag`, `Location`, `Set-Cookie`.
HTTP methods: HTTP verbs: GET read, POST create, PUT replace, PATCH partial update, DELETE remove, HEAD headers only, OPTIONS capabilities (CORS/preflight).
HTTP over TLS: encrypts request/response; prevents eavesdropping/tampering.
Idempotent: GET/PUT/DELETE/HEAD/OPTIONS; POST/PATCH usually not.
JSON.parse / JSON.stringify: convert between string and object; `JSON.stringify(x, replacer, space)` for pretty-print.
Parse with response.json(): `const data = await resp.json()` if response body is JSON.
Request headers: metadata client sends; often used for auth/content negotiation.
Response headers: metadata server sends; caching, cookies, location, type.
scheme: URL shape: `scheme://host:port/path?query#hash`
Simple request: GET/POST/HEAD with only s+imple headers and `Content-Type` in `x-www-form-urlencoded|multipart/form-data|text/plain`.
Status codes: HTTP codes: `200 OK`, `201 Created` (new resource, often `Location`), `204 No Content`, `304 Not Modified`, `400`, `401`, `403`, `404`, `409`, `413`, `422`, `429`, `500` (know what each implies for client/server behavior).
response.ok / response.status: check success; `ok` is true for 2xx.
Check response.ok / response.status: `if (!resp.ok) throw new Error(...)`.

JavaScript

Declarative focuses on the what: use array HOFs and state-driven UI; less manual loops.
Deep (structured): `structuredClone(obj)` deep clones supported types; JSON clone loses functions/Date/undefined.
Function declarations hoist: declarations hoist; function expressions/arrow functions do not.
Hoisting: declarations available earlier in scope; beware temporal dead zone for `let/const`.
Nullish coalescing vs OR: `a ?? b` only falls back on `null`/`undefined`; `a || b` falls back on any falsy (`0`, `""`, `false`).
Object.assign: shallow merge/copy: `Object.assign({}, a, b)`; nested refs remain shared.
Promises microtasks and await: promise callbacks run as microtasks after current stack; `await` pauses inside async fn.
Shallow copy: `{...obj}`, `[...arr]` copies top-level only; nested objects share refs.
Shallow copy pitfall: nested mutation affects both original and copy; must copy nested levels too.
StructuredClone caveat: only supported types; DOM nodes/functions not cloneable.
map/filter/reduce: map transforms, filter selects, reduce aggregates; prefer for clarity.
Array HOFs: Array HOFs: `map` transform (same length), `filter` keep subset, `reduce` aggregate, `some`/`every` booleans, `find` first match, `flatMap` map then flatten 1 level.
try/catch with await: wrap `await fetch(...)` to catch network/parse exceptions.
Event loop: call stack empties, then microtasks (promises), then task queue (timers/events).
Microtasks: promise `.then` callbacks run before timers after stack clears.
Function expression: not hoisted like declarations; can't call before definition.
Arrow function: lexical `this`; not hoisted like declaration.
spread operator: `...` expands iterable/object; used for shallow copy and merges.
Destructuring: `const {a} = obj`, `const [x] = arr` for concise extraction.
Template literals: `` `${x}` `` string interpolation; multiline strings.
Truthy/falsy: `0`, `""`, `null`, `undefined`, `false`, `NaN` are falsy; beware `||`.
find: returns first matching element or `undefined`.
flatMap: map then flatten one level; helpful for nested arrays.
reduce: fold list into accumulator; always define initial value when possible.
Promises: represent eventual value; resolve/reject once; chain with `.then/.catch`.
Promise.all: runs in parallel; rejects if any reject; returns array of results.
Promise.race: first settle wins; useful for timeouts.
Timeout pattern: race fetch with timeout promise for UI responsiveness.
Function parameter default: `function f(x=0){...}`.
Mutability: arrays/objects are mutable; primitives immutable.
Reference vs value: objects pass by reference value (pointer-like), primitives by value.

React Web

Controlled inputs: React state is source of truth; `value={x}` and `onChange={e=>setX(e.target.value)}`.
Data fetching pattern: use `useEffect(asyncFn)` style with inner async and cleanup/guards; set loading/error state.
Dependency array: `[]` mount, `[x]` when x changes, omitted runs every render (usually bug).
Functional component: returns JSX with single root; uses `className` not `class`.
Functional updater: `setX(prev => prev + 1)` when next state depends on previous.
Props: read-only inputs; destructure params; `props.children` for nested content.
React.memo: skips re-render if props shallow-equal; still re-renders on state/context changes.
Render and commit model: set state schedules render (VDOM diff) then commit updates DOM, then effects run.

Routers: BrowserRouter normal, HashRouter static hosts, MemoryRouter tests.
Routes and Outlet: nested routes render child with `<Outlet/>` in parent.
useNavigate: imperative navigation; `navigate("/home")`.
useParams: read `:id` segments; e.g., `/users/:id`.
useSearchParams: read/modify query string; like `?page=2`.
useRef: mutable `.current` without re-render; DOM nodes, timers, uncontrolled form reads.
useContext: read nearest Provider value; avoids prop drilling.
useMemo/useCallback: memoize value/function by deps; avoid recompute/recreate; beware stale closures.
Cleanup in useEffect: return function to unsubscribe timers/listeners; prevents leaks.
Effects run after commit: `useEffect` runs after paint; do side effects there.
Keys: stable unique key per sibling; avoid index if list can reorder.
Link/NavLink: declarative navigation; NavLink can style active route.

Storage and Authentication

Cookies: server sets via `Set-Cookie`; sent automatically with requests matching domain/path.
JWT: signed token; often stored in HttpOnly cookie to reduce XSS theft; still consider CSRF.
localStorage / sessionStorage: string-only; use JSON stringify/parse; writes do not trigger React re-render.
Mobile auth storage: prefer secure storage (keychain/keystore) for tokens; avoid plaintext async storage for secrets.

UX, Accessibility, IA

5 Es: effective, efficient, engaging, error tolerant, easy to learn.
Affordances: true supports action; hidden not apparent; false looks clickable but isn't.
Affinity Diagramming: cluster notes to themes to insights (feeds Define).
Assistive tech awareness: know implications for focus, semantics, labels, navigation.
Cognitive Walkthrough: per step ask: goal, see control, recognize it, get feedback.
Contextual Inquiry: observe in environment; master–apprentice interview; collect artifacts; derive requirements.
Design Thinking stages: Empathize, Define, Ideate, Prototype, Test, Implement.
Dialogue principles (ISO 9241-110): suitability for task, self-descriptiveness, controllability, expectations, error tolerance, individualization, learning.
Doherty Threshold (~400ms): keep interactions under ~0.4s or show progress to maintain flow.
Error Handling & Microcopy: prevent errors; when errors occur say what/why/how fix; preserve entered data.
Formative vs summative: formative is iterative during design; summative is final evaluation.
Formative testing: done during design to improve; smaller samples, qualitative.
Summative testing: final validation; can be more quantitative/benchmark-like.
Gestalt: proximity, similarity, continuity, closure; use for grouping/scan paths.
Information Architecture models: hub-and-spoke, fully connected, multi-level with breadcrumbs, wizard/stepwise, pyramid, pan-and-zoom, flat, modal panel, escape hatch.
Interaction paradigms: implementation-centric (direct functions), metaphoric (real-world analogy), idiomatic (learned conventions).
ISO 9241-11 usability: effectiveness, efficiency, satisfaction for specified users, tasks, contexts (use this wording).
Navigation principles: wayfinding aids, minimize nav cost (steps/switches/delays), provide global/utility/associative nav.
Pagination vs Infinite Scroll: paginate for discrete/findability/return-to-place; infinite for continuous feeds; watch footer/item finding.
Storyboarding: panels show user + context + goal across time; clarifies flows.
Universal Design principles: equitable use, flexibility, simple/intuitive, perceptible info, tolerance for error, low effort, size/space.
Von Restorff (Isolation) Effect: highlight one key thing with contrast/motion (judiciously).
WCAG A/AA/AAA: conformance levels; AA is common target.
Impairment types & time scales: sensory/motor/cognitive; permanent/temporary/situational; use for design justification.
Empty states and feedback: guide first use; show success and system status (loading/saving/queued).
Field testing: test in real environment; higher ecological validity, less control.
Laboratory testing: controlled environment; easier observation/measurement, less realistic.
Severity ratings: rank issues by impact/frequency/persistence; helps prioritize fixes.
Test plan: goals, tasks, participants, method, measures, script, logistics, analysis plan.
Moderator: runs session, asks questions, keeps neutral.
Why: in study reporting, "why" captures motivation/causality behind behavior (not just what happened). **What**: captures actions/outcomes users performed. **How**: captures process/strategy users used. **Who**: captures participant characteristics that matter. **Where**: captures context of use/environment. **Walkthrough Q1**: will user try to achieve the right effect/goal? **Walkthrough Q2**: will user notice the correct action/control is available? **Walkthrough Q3**: will user associate the correct action with the effect they want **Walkthrough Q4**: after acting, will user understand feedback shows progress?
User control and freedom: easy undo/redo; escape unwanted states.
Consistency and standards: follow platform norms; don't surprise users.
Error prevention: design to prevent mistakes before they happen.
Recognition rather than recall: keep options visible; reduce memory load.
Flexibility and efficiency of use: shortcuts for experts; efficiency without blocking novices.
Aesthetic and minimalist design: avoid irrelevant info; reduce clutter.
Error recovery: clear messages and recovery paths; preserve work.
Help and documentation: provide discoverable help when needed.

React Native and Mobile UI

Button: built-in simple button; limited styling.
Cards: flat, rectangular containers; good for previews/summary items.
Carousels: horizontal scrolling list of content cards; good for browseable sets.
Dimensions: read device width/height; use for responsive layout decisions.
FlatList: performant list for large/scrolling datasets; virtualizes rows.
Lists and grids: choose list for scan/ordering, grid for visual browsing; use consistent spacing.
Pressable: flexible touch wrapper; supports pressed/hover/focus styles and callbacks.
ScrollView: renders all children; fine for small lists, not huge datasets.
Stacks: vertical flows of screens; aligns with navigation stacks and stepwise tasks.
Swimlanes: horizontal lists inside vertical feed; good for categorized browsing.
VoiceOver/TalkBack: iOS/Android screen readers; ensure labels, roles, focus order.
Microinteractions: small feedback moments (tap, loading, success); improve clarity/feel if consistent.
Microinteraction scope: define trigger, rules, feedback, loops/modes; keep tight and consistent.
Direct manipulation: touch gestures (drag/pinch/scroll) map to visible changes; feels intuitive when consistent.
Accessibility grouping: group related controls/labels so SR navigation and scanning are coherent.

React Navigation

NavigationContainer: required root provider for navigation state.
Navigator rule: create navigator objects once (outside component) to avoid re-creating stacks.
Stack / Tab / Drawer: common navigation models; choose based on hierarchy vs peer sections vs global nav.

Mobile Gestures, Sensors, Deployment

Animated components: use Animated versions like `Animated.View`, `Animated.Text`, `Animated.Image`.
Animated math: use `Animated.add` etc, not `+`, when composing animated values.
Animated requirement: define what animates and how (timing/spring) before coding.
Drag: gesture where user drags an object; often uses pan responder or gesture handler.
Double-tap: double click equivalent; common for zoom/like.
expo-sensors: sensors API; permissions and device support vary by platform.
Gestures: tap, double-tap, long-press, swipe, pinch, spread; map to clear feedback.
gesture-handler: higher-level gesture recognition than low-level responders.
Long-press: press and hold; often opens context menu or alternate action.
Multi-touch: recognizing two+ fingers; needed for pinch/spread.
PanResponder: low-level gesture handling; provides dx/dy, grant/move/release callbacks.
Permissions: required for some sensors/APIs; request and handle denial gracefully.
Sensor availability: not all devices support all sensors; always feature-detect & provide fallbacks.

Common sensors: Accelerometer, Barometer, Gyroscope, LightSensor, Magnetometer, Pedometer.
EAS: Expo Application Services for builds/submission; replaces local native build complexity.
APK: Android package; installable build artifact.
IPA: iOS app archive; used for TestFlight/App Store distribution.
Deployment: packaging, signing, store submission, monitoring; differs iOS vs Android.
App review: iOS App Store review is stricter; ensure privacy/permissions justification and stability.
Monitoring: track crashes/perf; add logging and error reporting in production.
Performance: watch overdraw, large lists, unnecessary re-renders; use FlatList, memoization.
expo-camera: camera access; permission required; handle denial.
expo-av: audio/video playback; manage lifecycle and interruptions.
expo-haptics: haptic feedback for microinteractions.
expo-brightness: adjust screen brightness; permission and UX caution.
expo-battery: battery info; consider power impact.

Prototyping

Annotations: add intent notes to wireframes; explains behavior not visible in static sketch.
Extreme prototyping: static → interactive → real; refine by fidelity stages.
Fidelity principle: higher fidelity narrows feedback to look/feel; lower fidelity reveals structure/concept issues.
Horizontal prototype: broad coverage, shallow depth; explores IA and navigation.
Vertical prototype: deep slice of one flow; tests feasibility/interaction details.
Paper prototyping: sketch UI and simulate interaction; fast iteration early.
Evolutionary: prototype becomes product through refinement.
Incremental: build slices/features then integrate gradually.
Wireframes: low-fidelity layout/structure; not visual polish.
Experience prototyping: prototype the experience context, not just screens; act out to learn.
Bodystorming: role-play in real space; find issues in flow/context.
Define context: specify user, environment, constraints, goals before prototyping.
Develop scenarios: write realistic stories/tasks that drive prototype interactions.
Identify design goals: define what questions you want prototype to answer.
Set up environment: recreate context (props, space, timing) for realistic behavior.
Act out interaction: simulate user/system actions; capture breakdowns.
Develop insight: synthesize observations into requirements/design changes.
Role: in experience prototyping, assign roles (user, system, bystander) to surface context issues.
Implementation: prototype what is implemented (logic) vs what is only represented
Integration: prototype how components connect (handoffs/data flow), not just individual screens.
Look and feel: prototype visual/interaction tone; helps emotional response testing.

Conversational Interfaces

Acknowledgements: short confirmations like “Got it”, “Okay”; reduce uncertainty.
Agent: the system/entity user interacts with (assistant, bot, device).
Command-and-control: user issues direct commands; system executes; needs clear confirmations/errors.
Conversational interface technology: SR, NLU, dialog management, response generation, TTS.
Conversational error handling: handle no-input, misrecognition, wrong intent, ambiguous entities; offer repair.
Dialog management: controls conversation state and next system action.
Entity: extracted parameter/value (date, location, item).
Explicit confirmation: ask user to confirm (“Did you mean X?”).
Implicit confirmation: reflect back in action (“Okay, setting alarm for 7”).
Flowcharting: map states, intents, errors, confirmations; prevents missing branches.
Gricean maxims: quality, quantity, relevance, manner; guide cooperative responses.
Interruption: user interrupts agent; need barge-in handling and state recovery.
Message: unit of dialogue (user or assistant); can be multimodal.
Multimodality: combine voice, text, visuals; choose channel that reduces ambiguity.
No speech detected: no-input error; reprompt with guidance and maybe alternative input mode.
Positive feedback: praise/affirmation (“Nice!”) used carefully; can encourage continued interaction.
Question: prompt type; open vs closed affects user effort and ambiguity.
Recognized but wrong action: system heard speech but mapped wrong intent; provide repair and clarify options.
Response generation: produces phrasing/structure of reply based on state and data.
Speech recognition: audio to text; impacted by noise, accents, microphones.
Spoken language understanding: parse intent/entities from text; may be probabilistic.
Text-to-speech: convert response text to audio; voice/tone matters.
Timeline markers: words like “first”, “then”, “next”, “after that”; help multi-step guidance.
Turn-taking: conversational control of who speaks; needs cues and timing.
Wake word/button: activation mechanism; reduces accidental triggers; sets user expectation.
Wit.ai response: includes intents with confidence, entities, and text; use confidence thresholds to decide confirmation.
Conversational interface: natural language for interaction; must handle ambiguity and repair.

Social and Personality Design

Big Five: personality traits OCEAN; traits not types; useful for tone/persona adaptation.
CASA: Computers Are Social Actors; users apply social rules to tech.
Consistency-attraction: users like agents that behave consistently over time.
External consistency: consistent with other systems/users’ expectations; reduces learning burden.
Face saving: help user avoid embarrassment; soften corrections.
Face threatening: responses that embarrass or blame; avoid.
Internal consistency: consistent behavior/tone within your agent; builds trust.
Mindlessness: users respond automatically to cues (voice, politeness) without deep analysis.
Myers-Briggs: personality “types” model; less scientific than Big Five; use carefully if at all.
Negative face: desire for autonomy; respect it with options and non-pushy phrasing.
Negative politeness: indirect, respectful phrasing; reduces imposition (“If you’d like…”).
Pareidolia: humans see patterns/intent in randomness; can anthropomorphize agents.
Persona development: define agent values, tone, vocabulary, boundaries, and capabilities.
Persona-level matching: adapt style to user preference/context; avoid uncanny mimicry.
Politeness theory: positive vs negative face; choose strategies per context.
Positive face: desire to be liked/approved; reinforce with respectful, affirming language.
Positive politeness: friendly solidarity cues; can increase warmth/trust.
Similarity-attraction: users prefer agents that feel similar to them (language/tone/values).
Similarity matching: align phrasing and level of detail to user style without copying.
Expressing expertise: show competence with clear reasoning, limits; avoid overconfidence.

Generative AI and Streaming

Roles: platform/system, developer, user, assistant; roles influence instruction priority and behavior.
react-markdown: render Markdown output safely in React; useful for assistant responses.
ReadableStream: streaming fetch body; read chunks progressively for “typing” UI.
TextDecoder: decode bytes to text while streaming; handle partial multibyte chars.
Chunk parsing: TCP/stream chunks can split JSON; buffer carryover until full JSON object forms.

Backend and Deployment

Docker build: builds a container image from a Dockerfile.
Docker run: runs a container from an image.
Docker stop: stops a running container.
Docker rm: removes a stopped container.
Express endpoint: define route with callback `(req, res) => { ... }`.
Express middleware: function that can read/modify req/res then call `next()`; runs in order.
req vs res: request data in `req` (params/body/query), send output with `res`.
res not return: Express sends via `res.status(...).json(...)`, not `return value`.
SQL queries: interact with DB using query strings (prefer parameterized queries).
SQLite helpers: common helpers `exec`, `run`, `get`, `all` for running SQL.