CSS Selectors & Specificity

- Basic: `tag`, `.class`, `#id`, `tag.class`, `[attr=value]`, `A B` (descendant), `A > B` (child), `A + B` (adjacent), `A ~ B` (siblings), `*`.
- Specificity order (highest → lowest): inline style > `#id` > class/attr/pseudo-class > tag/pseudo-element > `*`.
- Common patterns: `.btn.primary`, `ul > li.active`, `a[href^="/"]`, `input:focus`

HTTP Methods & Semantics

- **GET** (read), **POST** (create), **PUT** (replace), **PATCH** (partial update), **DELETE** (remove), **HEAD** (headers only), **OPTIONS** (capabilities).
- **Idempotent:** GET/PUT/DELETE/HEAD/OPTIONS. **Non-idempotent:** POST/PATCH (usually).
- **200 OK**, **201 Created** (new resource; include `Location`), **204 No Content** (success, no body). **304 Not Modified** (cache validation). **400 Bad Request** (client input invalid), **401 Unauthorized** (needs auth), **403 Forbidden** (authenticated but disallowed), **404 Not Found**. **409 Conflict** (versioning/duplicate), **413 Payload Too Large**, **422 Unprocessable Entity** (valid JSON, semantically invalid), **429** is too many requests. **500 Server Error**.
- `scheme://host:port/path?query#hash`
- Use **path** to identify the resource (`/users/123`), **query** to filter/sort/paginate (`?q=alice&page=2`).

Headers You'll Use

- Request: `Accept`, `Authorization: Bearer <token>`, `Content-Type: application/json`, custom like `X-CS571-ID`.
- Response: `Content-Type`, `Cache-Control`, `ETag`, `Location`, `Set-Cookie`.
- **Simple request:** method GET/POST/HEAD; only simple headers; `Content-Type` must be `application/x-www-form-urlencoded` | `multipart/form-data` | `text/plain`.
- Otherwise browser sends **preflight** `OPTIONS` with `Access-Control-Request-Method/Headers`.
- Server must reply with `Access-Control-Allow-Origin`, `Allow-Methods`, `Allow-Headers`, maybe `Allow-Credentials`.
- Client sends `If-None-Match: <etag>` or `If-Modified-Since`. If unchanged → **304**.
- Server controls caching with `Cache-Control: max-age=...`, `no-store`, `must-revalidate`.

- HTTP over TLS: encrypts request/response in transit; prevents eavesdropping/tampering.
- **Types:** string, number, boolean, null, array, object. **No comments, no trailing commas.**
- Parse/stringify with `JSON.parse` / `JSON.stringify` (replacer/space optional).
- If API needs form data: use `FormData` (auto sets proper `Content-Type`; don't set it manually).

Values & Equality

- Prefer `===`/`!==`. Avoid `==` coercion traps.
- `nullish coalescing (??)` only replaces **null/undefined**; `||` treats `0`, `''`, `false` as falsy.

Variables & Hoisting

- `let`/`const` (block scope). `const` prevents rebinding, **not** mutation.
- Function declarations hoist; function expressions/arrow do **not**.

Copying & Mutation

- **Shallow copy**: `{...obj}`, `[...arr]`, `Object.assign`.
- **Deep (structured)**: `structuredClone(obj)` (only supported types). JSON trick loses functions/undefined/Date.
- Pitfall: shallow copy keeps nested references—mutating nested objects mutates both.

Arrays (HOFs)

- `map` (same length, transform), `filter` (subset), `reduce` (aggregate), `some`/`every` (boolean checks), `find` (first match), `flatMap`.
- Declarative focuses on the what (use the easy maps). Imperative is the how (more technical)

Async

- Promises microtasks run after the current stack; `await` pauses within async function.
- Pattern: check `response.ok`, handle errors, parse JSON, wrap in try/catch.

DOM:

- Query: `document.querySelector(css)`, `querySelectorAll`, `getElementById`.
- Text vs HTML: `.textContent` (safe) vs `.innerHTML` (parses HTML; XSS risk).
- Classes: `el.classList.add/remove/toggle`.
- Events: pass a **function reference** to `addEventListener`.
- **Bubbling/Capture:** optional `{ capture: true }`; default is bubble.
- Control flow: `event.preventDefault()`, `event.stopPropagation()`.
- Forms: call `preventDefault()` on submit before async work.

Fetch API (glue to servers)

- Configure method, headers (e.g., `Content-Type: application/json`, `X-CS571-ID`), and body (stringified JSON as needed).
- Check `response.ok` / `response.status`; parse with `response.json()` when applicable.
- **Credentials & cookies:** set `credentials: 'include'` for cross-site cookies (server must allow via CORS).
- A fetch doesn't update UI unless you set state or update the DOM.

Components & Props

- Functional component returns JSX (single root). Use `className`, not `class`.
- Props are read-only. Destructure in params. `props.children` for nested content.
- Lists: add **stable** `key` per sibling (avoid index if order may change).

State (useState)

- Returns pair of value and setter. Updates are **async & batched**.
- Prefer **functional updater** when next state depends on previous.
- **Replace, don't merge:** setting object/array **replaces** it; use spread to keep fields (shallow).
- **Never call `setState` during render** (causes infinite re-render). Do it in events/effects.

Effects (useEffect)

- Runs **after commit**. Dependencies control when: `[]` (mount), `[x]` (when `x` changes), omitted (every render—usually a bug).
- **Cleanup** subscriptions/timeouts in the returned function.
- **Data fetching pattern:** guard against setting state after unmount; handle errors; include relevant deps.

Refs, Context, Memoization

- **useRef:** mutable `.current` that **does not** trigger re-render; good for timers, DOM nodes, uncontrolled inputs.
- **useContext:** read value from nearest `<Context.Provider>`.
- **useCallback / useMemo:** memoize function/value by deps; avoid re-creating handlers/expensive recompute. Beware **stale closures** (empty deps capture initial values).
- **React.memo:** skip re-render if props shallow-equal.

Render/Commit Mental Model

- Set state → React schedules render (virtual DOM diff) → **commit** updates to real DOM → effects run. Console logs in render vs `useEffect` vs promise callbacks occur at different times.

React Router (SPA)

- Routers: `BrowserRouter` (normal), `HashRouter` (static hosts), `MemoryRouter` (tests).
- Routes: define paths, nest routes, and render child route with `<Outlet/>` in parent.
- Navigation: use `<Link>` / `<NavLink>` for declarative nav; `useNavigate()` for imperative nav.
- Params & search: `useParams()` for `:id` segments; `useSearchParams()` for query strings.
- **Controlled inputs**: value in React state; `onChange` updates state. Pros: validation, instant UI, single source of truth. Cons: more renders. **Uncontrolled** inputs: DOM holds value; read via `ref`. Pros: fewer renders, simpler small forms.

Storage, Cookies, Auth

- **localStorage / sessionStorage:** string-only; use JSON stringify/parse as needed. Writes don't auto-trigger re-renders.
- **Cookies (server sets via `Set-Cookie`):**
  - Flags: **HttpOnly** (JS can't read), **Secure** (HTTPS only), **SameSite** (Lax/Strict/None; `None` requires Secure).
  - Cross-site cookie use often needs `credentials: 'include'` on fetch plus CORS `Allow-Credentials`.
- **JWT:** often stored in **HttpOnly** cookie to mitigate XSS token theft; still consider **CSRF** (use SameSite and/or CSRF tokens).
- **Credentialed requests:** remember both client option and server CORS settings.

Design & HCI (what gets asked)

- **Heuristic Evaluation (Nielsen 10):** system status; match to real world; user control/freedom; consistency/standards; error prevention; recognition over recall; flexibility/efficiency; minimalist design; error recovery; help/docs.
- **Cognitive Walkthrough:** for each step—will user form right goal, see right control, recognize it, and get feedback?
- **Think-Aloud:** users verbalize thoughts; detect confusion points.
- **Contextual Inquiry:** observe in users' environment; master–apprentice interview; collect artifacts; feed requirements.
- **Affinity Diagramming:** cluster notes → themes → insights (supports **Define**).
- **Storyboarding:** panels showing user + context + goal across time (flows).
- **Design Thinking stages:** Empathize → Define → Ideate → Prototype → Test → Implement.
- **Interaction paradigms:** Implementation-centric (direct functions), Metaphoric (real-world analogy), Idiomatic (learned UI conventions).
- **Affordances:** true (supports action), hidden (not apparent), false (looks clickable but isn't).
- **Navigation principles:** wayfinding aids, minimize cost (steps/switches/delays), provide global/utility/associative nav.
- **Navigation models:** hub-and-spoke; fully connected; multi-level (breadcrumbs); stepwise/wizard; pyramid; pan-and-zoom; flat; modal panel; clear entry points; bookmarks; **escape hatch**.
- **Pagination vs Infinite Scroll:** paginate when discrete results/findability/returning to place matters; infinite for continuous feeds; note drawbacks (footer access, locating items).
- **Visual scanning: F-pattern** (text-heavy), **Z-pattern** (simple hero pages). Use contrast/hierarchy/focal point.
- **Gestalt:** proximity, similarity, continuity, closure.
- **WIMP:** Windows, Icons, Menus, Pointer.
- **Focal point:** strongest visual attractor guiding initial attention.
- **Labels:** `<label htmlFor="id">` with matching `id` on input.
- **Headings:** logical order (`h1`→`h2`→`h3`).
- **Contrast:** meet WCAG contrast; don't rely on color alone (add text/icons/patterns).
- **Keyboard:** everything reachable/actionable by keyboard; logical tab order; **visible focus**.
- **Alt text:** describe function/meaning.
- **Landmarks/roles:** `<main>`, `<nav>`, `<header>`, `<footer>`, `role="dialog"` with focus management.
- **Form errors:** connect messages with `aria-describedby`, set `aria-invalid="true"` when invalid.