QC 3

Web Development Study Guide

---

1. HTTP & Web Fundamentals

1.1 HTTP Request Lifecycle

Be able to describe step-by-step what happens when a user types a URL and hits Enter:

1. **URL Parsing**
   - Browser parses the URL into protocol, domain, port (optional), path, and query string.
2. **DNS Resolution**
   - Browser checks cache or queries a DNS server to resolve the domain to an IP address.
3. **TCP (and TLS) Connection**
   - Browser opens a TCP connection to the server at that IP on the appropriate port (80 for HTTP, 443 for HTTPS).
   - For HTTPS, performs the TLS handshake to establish an encrypted channel.
4. **HTTP Request**
   - Browser sends an HTTP request consisting of:
     - Request line: method + path + HTTP version (e.g., `GET /home HTTP/1.1`)
     - Headers: `Host`, `User-Agent`, `Accept`, `Cookie`, etc.
     - Optional body (for methods like POST, PUT, PATCH).
5. **Server Handling**
   - Server receives the request and routes it (via a web server or application framework).
   - Application executes logic: authentication, fetching/updating data from a database, applying business rules.
6. **HTTP Response**
   - Server returns:
     - Status line: e.g., `HTTP/1.1 200 OK`
     - Headers: `Content-Type`, `Set-Cookie`, `Cache-Control`, etc.
     - Body: often HTML, JSON, or other data.
7. **Browser Rendering**
   - Browser parses the HTML and builds the **DOM**.
   - Parses CSS and builds the **CSSOM**.
   - Combines DOM + CSSOM → **Render Tree**.
   - Performs layout and painting to display the page.
   - Executes JavaScript, which can further modify the DOM.

    - As HTML is parsed, browser issues additional HTTP requests for CSS, JS, images, fonts, etc.

1.2 Statelessness and Cookies

Stateless HTTP:

- Each HTTP request is independent.
- The server does not remember past requests by default.
- There is no built-in concept of "logged-in user" or "session" between requests.

How Cookies bridge the gap:

- A **cookie** is a small key–value pair stored in the browser.
- Server sets cookies using the `Set-Cookie` response header.
- Browser stores them and sends them back on each request via the `Cookie` header.
- Cookies often contain:
    - Session IDs (which the server maps to user state).
    - Preference information.
    - Security tokens (carefully handled).
- Cookie attributes:
    - `Secure`: only sent over HTTPS.
    - `HttpOnly`: not accessible via JavaScript (protects from XSS).
    - `SameSite`: controls cross-site sending behavior.

1.3 HTTP Methods

GET vs POST (beyond "retrieve" vs "send"):

- **Data visibility**
    - GET: sends data in the URL (path + query string). Visible in the address bar, logs, and browser history.
    - POST: sends data in the request body. Not visible in the URL.
- **Caching**
    - GET: generally cacheable by default; browsers and proxies may cache responses.
    - POST: not cacheable by default.
- **Idempotency / Semantics**
    - GET: should be **safe** (no side effects) and **idempotent**.
    - POST: not necessarily idempotent (repeated POSTs can create multiple resources).

PUT vs PATCH:

- **PUT**
  - Intended as a **full replacement** of the resource.
  - Client sends the entire representation.
  - Often treated as idempotent.
- **PATCH**
  - Intended for **partial update**.
  - Payload describes only the changes (e.g., only one field updated).
  - Often used when sending JSON "patch" operations or partial user updates.

1.4 HTTP Headers: Content-Type vs Accept

- **Content-Type**
  - Describes the format of the **body being sent** in the request/response.
  - Example (request): `Content-Type: application/json` means "the body I'm sending is JSON."
  - Example (response): `Content-Type: text/html` means "the body you're receiving is HTML."
- **Accept**
  - Sent by the **client** to say what formats it can handle in the response.
  - Example: `Accept: application/json, text/html`.

1.5 HTTP Status Codes

4xx vs 5xx Errors:

- **4xx – Client errors**
  - The request is bad from the client's side.
  - Examples:
    - 401 Unauthorized: client is not authenticated (e.g., missing or invalid token).
    - 403 Forbidden: client is authenticated but not allowed to access that resource.
    - 404 Not Found: server cannot find the requested resource.
- **5xx – Server errors**
  - The request was valid, but the server failed to process it.
  - Example:
    - 500 Internal Server Error: an unexpected error occurred on the server (uncaught exception, DB failure, etc.).

3xx – Redirects:

- 3xx codes indicate the client should use a different URL.
- **301 Moved Permanently**
  - Resource is permanently moved to a new URL.
  - Browsers and search engines update links and cache the new location.
- **302 Found / Temporary Redirect**
  - Resource is temporarily located at another URL.
  - Clients should keep using the original URL in the future.

---

2. HTML Structure & Semantics

2.1 DOM Tree and HTML

- HTML is text; the browser parses it into a **DOM (Document Object Model)**.
- The DOM is a **tree of nodes** in memory:
  - Document → `<html>` → `<head>` and `<body>` → nested elements → text nodes.
- JavaScript interacts with the DOM using APIs like:
  - `document.getElementById`, `querySelector`, `appendChild`, etc.
- Changes to the DOM in JavaScript immediately affect the rendered page.

2.2 Block vs Inline Elements

- **Block-level elements:**
  - Start on a new line.
  - Expand to fill the available horizontal space.
  - Respect width/height and vertical margins.
  - Examples: `<div>`, `<p>`, `<h1>`–`<h6>`, `<section>`.
- **Inline elements:**
  - Flow within a line of text.
  - Only as wide as their content.
  - Width/height properties generally don't apply in the same way.
  - Examples: `<span>`, `<a>`, `<strong>`, `<em>`.
- **Containment:**
  - Semantically, inline elements should not contain block-level elements (though some browsers may try to fix invalid markup).

2.3 Semantic HTML

- Semantic elements convey **meaning**, not just presentation:
  - `<header>`, `<nav>`, `<main>`, `<article>`, `<section>`, `<aside>`, `<footer>`.
- Benefits:
  - **Accessibility**:
    - Screen readers can jump between landmarks (e.g., "skip to main content").
    - Clearer structure for assistive technologies.
  - **SEO**:
    - Search engines better understand what content is important.
  - **Maintainability**:
    - Easier for developers to read and reason about structure.

2.4 Forms

Server-side validation (even with HTML5 validation):

- HTML5 validation (`required`, `pattern`, `type="email"`, etc.) is useful for user experience.
- But front-end validation can be bypassed:
  - Disabled JavaScript.
  - Custom HTTP clients.
- Server-side validation is necessary for:
  - Security.
  - Data integrity.
  - Consistency across clients.

Associating `<label>` and `<input>`:

- Correct association:
  - Using `for` and `id`:
  - `<label for="email">Email</label>`
  - `<input id="email" name="email" type="email">`
  - Or wrapping the input:
  - `<label>`
  - `  Email`
  - `  <input name="email" type="email">`
  - `</label>`
- Why it matters:
  - Clicking the label focuses the input → better usability.
  - Screen readers announce the label with the input → better accessibility.

HTML5 input types and mobile browsers:

- Special input types:
  - `type="email"`, `type="tel"`, `type="number"`, `type="date"`, etc.
- Mobile browsers:
  - Show optimized keyboards (e.g., numeric keypad for `number`, email keyboard for `email`).
  - May show native pickers (e.g., date picker).
- Desktop browsers:
  - Same validation semantics; UI enhancements depend on the browser (e.g., spinners, date widgets).

---

3. CSS Styling & Layout

3.1 Box Model

- Each element's box includes:
  - **Content → Padding → Border → Margin**.
- Default (`box-sizing: content-box`):
  - Total width = width + left/right padding + left/right border (plus margin outside that).
- Example:
  - `width: 200px; padding: 20px; border: 0;`
  - Total width = 200 + 20 + 20 = 240px (plus margins).
- `box-sizing: border-box`:
  - Sets the **total** width to the declared width (including content, padding, border).
  - Same example with `box-sizing: border-box`:
    - Total width stays 200px (content shrinks to accommodate padding).

3.2 Specificity & Cascade

- CSS specificity hierarchy:
  1. Inline styles (e.g., `style="..."`) – highest.
  2. ID selectors (`#id`).
  3. Class/attribute/pseudo-class selectors (`.class`, `[attr]`, `:hover`).
  4. Element/pseudo-element selectors (`div`, `p`, `::before`).
- If an element is targeted by ID, class, and element selectors:
  - The ID selector wins.
  - If two rules have the same specificity, the one that appears **later** in the stylesheet wins.

3.3 Layout Systems: Flexbox vs Grid

- **Flexbox**:
  - One-dimensional layout (either row or column).
  - Best for:
    - Navbars.
    - Toolbars.
    - Aligning items in a single row/column.
    - Simple responsive layouts.
- **Grid**:
  - Two-dimensional layout (rows and columns).
  - Best for:
    - Complex page layouts.
    - Dashboards.
    - Photo galleries.
    - Any layout where positioning in both directions matters.

3.4 Positioning

- `position: relative` on a parent:
  - Keeps the element in normal document flow.
  - Establishes a new containing block for absolutely positioned descendants.
- `position: absolute` on a child:
  - Removes the element from normal document flow.
  - Positioned relative to the **nearest ancestor** with a non-static position (like `relative`).
  - Uses `top`, `left`, `right`, `bottom` offsets.

3.5 Responsive Design & Media Queries

- A responsive site:
  - Adapts layout and design to different screen sizes.
  - Uses flexible units (%, `rem`, `vw`, `vh`).
  - Uses media queries to change styles at certain breakpoints.
- Media queries:
- `@media (max-width: 768px) {`
- `  .nav {`
- `    flex-direction: column;`
- `  }`
- `}`
- Mobile-first approach:
  - Base styles for small screens.

- Add `@media (min-width: ...)` rules for larger screens.

3.6 CSS Variables

- Declaring:
- `:root {`
- `  --main-color: #3498db;`
- `}`
- Using:
- `button {`
- `  color: var(--main-color);`
- `}`
- Benefits:
  - Centralized theming.
  - Easy to override in specific components/containers.
  - Variables follow normal CSS cascade and inheritance.

3.7 Pseudo-classes vs Pseudo-elements

- **Pseudo-classes** (state selectors):
  - `:hover`, `:focus`, `:active`, `:nth-child()`, etc.
  - Represent a state of an existing element.
- **Pseudo-elements** (sub-part selectors):
  - `::before`, `::after`, `::first-line`, etc.
  - Represent a part of an element (like a virtual element for decorative content).

---

4. JavaScript Core & Logic

4.1 var, let, const and Scope

- `var`:
  - Function-scoped.
  - Hoisted and initialized with `undefined`.
  - Not block-scoped (ignores `{}` blocks except functions).
- `let` and `const`:
  - Block-scoped (limited to `{}`).
  - Hoisted but in the temporal dead zone until execution reaches the declaration line.
  - `const` cannot be reassigned.

4.2 Hoisting

- Hoisting is JavaScript's behavior of moving declarations to the top of their scope during compilation.
- **Function declarations**:
  - Fully hoisted.
  - Can be called before they appear in code.
- **Function expressions / arrow functions** with `const`/`let`:
  - The variable is hoisted but not initialized.
  - Accessing before declaration gives a `ReferenceError`.

4.3 `this` Context

- In regular functions:
  - `this` is determined by how the function is called:
    - Method call: `obj.method()` → `this` is `obj`.
    - Standalone: `func()` → `this` is `undefined` in strict mode (or `window` in non-strict).
    - Explicitly set: `func.call(obj)`, `func.apply(obj)`, `func.bind(obj)`.
- In arrow functions:
  - `this` is **lexically bound** to the surrounding scope.
  - Does not create its own `this`.

4.4 Type Coercion: == vs ===

- == (loose equality):
  - Performs type coercion if types differ.
  - Examples:
    - `1 == '1'` → true.
    - `0 == ''` → true.
    - `null == undefined` → true.
- === (strict equality):
  - No type coercion; both type and value must match.
  - Examples:
    - `1 === '1'` → false.
    - `null === undefined` → false.
- Unexpected cases often come from using ==; interviews usually expect "always prefer ===."

4.5 ES6+ Features

Spread vs Rest (. . .):

- Rest (collect):
- `function myFunc(...args) {`
- `  // args is an array of all arguments`
- `}`
- Spread (expand):
- `const newArr = [...oldArr, 4, 5];`
- `const newObj = { ...oldObj, extra: true };`

Template Literals:

- Use backticks:
- `const name = 'Sidney';`
- `const greeting = \`Hello, ${name}!\`;`
- Advantages:
    - Expression interpolation.
    - Multi-line strings without `\n`.
    - Tagged templates (advanced).

Default Parameters:

- Setting defaults:
- `function greet(name = 'Guest') {`
- `  console.log(\`Hello, ${name}\`);`
- `}`
- Behavior:
    - Passing `undefined` uses the default.
    - Passing `null` does NOT use the default (value is `null`).

4.6 Strict Mode

- `"use strict"`:
    - Enables stricter parsing and error handling.
    - Examples:
        - Disallows using undeclared variables.
        - Throws errors for assignments to non-writable properties.

- Changes `this` in simple function calls to `undefined` (instead of `window`).

---

5. JavaScript DOM & Asynchronous Programming

5.1 Event Handling

Event Bubbling vs Capturing:

- **Capturing phase**:
  - Event travels from the root (window/document) down to the target element.
- **Target phase**:
  - Event is at the target element.
- **Bubbling phase**:
  - Event travels from the target back up to the root.
- `addEventListener(type, handler, { capture: true })` listens in the capturing phase.
- By default (`capture: false`), listeners handle events in the bubbling phase.
- `event.stopPropagation()`:
  - Stops further propagation in the current phase.

Event Delegation:

- Attach a single listener to a parent element rather than many children.
- Example: one `click` listener on `<ul>` handles clicks on any `<li>`.
- Benefits:
  - Performance (fewer listeners).
  - Dynamically added children automatically work.

5.2 Asynchronous JavaScript

Promises vs Async/Await:

- Promises:
  - `fetch(url).then(...).catch(...).`
  - Can lead to nested chains.
- Async/await:
  - Syntactic sugar over Promises.
  - Makes async code look synchronous:
  - `async function loadData() {`
  - `  try {`

```
    const response = await fetch(url);
    const data = await response.json();
  } catch (err) {
    console.error(err);
  }
}
```

- Error handling with `try/catch` is more familiar and readable.

Fetch API and errors:

- `fetch()` returns a Promise that:
  - **Rejects** on network errors (no response).
  - **Resolves** even for HTTP errors like 404 or 500.
- You must check `response.ok` or `response.status` yourself:
- `const res = await fetch(url);`
- `if (!res.ok) {`
- `  // handle HTTP error`
- `}`

5.3 JSON Handling

- Valid JSON supports:
  - Objects, arrays, numbers, strings, booleans, `null`.
- Cannot represent:
  - Functions.
  - `undefined`.
  - Symbols.
- Use `JSON.stringify()` to send data; use `JSON.parse()` (or `response.json()`) to read it back.

---

6. TypeScript

6.1 TypeScript vs JavaScript

- TypeScript:
  - Superset of JavaScript adding static typing and type-checking.
- TypeScript compiler:
  - Checks types at build time (not runtime).
  - Compiles TS into plain JavaScript that browsers can run.
- Browsers do not run TypeScript natively.

6.2 `any` vs `unknown`

- `any`:
  - Opts out of type checking.
  - You can do anything with it (call it, access properties) without errors from TypeScript.
- `unknown`:
  - Safer alternative.
  - You must narrow it (e.g., using `typeof`, `instanceof`, or custom type guards) before using properties or methods.

Example:

```
let value: unknown;
if (typeof value === 'string') {
  value.toUpperCase(); // Now safe
}
```

6.3 Interfaces vs Type Aliases

- Both can describe object shapes:
- `interface User {`
- `  id: number;`
- `  name: string;`
- `}`
-
- `type UserAlias = {`
- `  id: number;`
- `  name: string;`
- `};`
- Prefer **interfaces** when:
  - You want declaration merging or extension.
  - You're designing public APIs that may be extended.
- Type aliases are more flexible:
  - Can represent unions, primitives, etc.

6.4 Union Types

- A union type allows a variable to be one of multiple types:
- `let id: string | number;`
- You must narrow the type before using it:
- `if (typeof id === 'string') {`
- `  // treat id as string`
- `}`

- Gives flexibility with type safety, unlike many class-based languages.

6.5 Tuples

- Tuple example:
- `let pair: [string, number];`
- `pair = ['age', 25];`
- Differences from `(string | number)[]`:
  - Tuples have fixed length and types at specific positions.
  - `(string | number)[]` can have elements in any order and any length.

6.6 Type Guards

- User-defined type guard:
- `function isUser(obj: any): obj is User {`
- `  return obj && typeof obj.name === 'string';`
- `}`
- In an `if (isUser(x))` block, TypeScript knows `x` is `User`.

6.7 Generics

- Generic array of strings:
- `const names: Array<string> = [];`
- Generics make functions reusable:
- `function identity<T>(value: T): T {`
- `  return value;`
- `}`
- Constraints with `extends`:
- `function logLength<T extends { length: number }>(value: T) {`
- `  console.log(value.length);`
- `}`

6.8 `keyof`, Mapped Types, Utility Types

- `keyof`:
- `interface User {`
- `  id: number;`
- `  name: string;`
- `}`
- `type UserKeys = keyof User; // 'id' | 'name'`

- Mapped types:
- `type ReadonlyUser = {`
- `  [K in keyof User]: Readonly<User[K]>;`
- `};`
- Utility types:
  - `Partial<T>`: all properties optional.
  - `Pick<T, K>`: only selected properties.
  - `Omit<T, K>`: exclude specific properties.
- Example use:
  - `Partial<User>` for update operations where not all fields are required.

6.9 `tsconfig.json` Target

- `target` specifies the JavaScript version the TS compiler outputs:
  - e.g., ES5, ES2015 (ES6), ES2020.
- Affects:
  - Syntax transforms (arrow functions, classes).
  - Feature availability and polyfills.
- Lower targets support older browsers but produce more verbose code.

---

7. React: Core Concepts & Components

7.1 Virtual DOM

- Virtual DOM:
  - An in-memory representation of the UI.
- On state changes:
  - React creates a new virtual DOM tree.
  - Diffs it with the previous tree.
  - Applies minimal real DOM updates (reconciliation).
- Benefits:
  - Fewer direct DOM operations.
  - Performance and predictability.

7.2 JSX

- JSX:
  - Syntax extension that looks like HTML in JavaScript.
  - Compiled to `React.createElement()` calls.

- Browsers cannot read JSX directly; build tools (Babel) transform it into JavaScript.

7.3 Props vs State

- Props:
  - Inputs from parent components.
  - Read-only from the child's perspective.
- State:
  - Internal data managed by the component.
  - Updated using `setState` (class) or `useState` (function).
  - Changes trigger re-renders.

7.4 Unidirectional Data Flow

- Data flows **down** from parents to children via props.
- Children notify parents via callback functions.
- No automatic two-way binding; this makes data flow predictable and easier to debug.

7.5 Controlled vs Uncontrolled Components

- Controlled:
  - Form input values are driven by React state.
  - `value={stateValue}`, `onChange` updates state.
  - React is the single source of truth.
- Uncontrolled:
  - Form values are stored in the DOM.
  - Access via refs when needed.
  - Less control; typically used for simple forms.

7.6 Keys in Lists

- Keys help React track items in a list across renders.
- Good keys: stable, unique identifiers from data.
- Using array index as key can cause:
  - Incorrect item reuse when order changes.
  - Input fields showing wrong data.
  - Animation and state mismatch issues.

7.7 Synthetic Events

- React wraps native events in a SyntheticEvent for:
  - Cross-browser consistency.
  - Performance optimizations.
- Behaves like native events but standardized.

---

8. React: Hooks, Lifecycle & State Management

8.1 Rules of Hooks

- Only call hooks:
  - At the top level of React function components or custom hooks.
  - Never inside loops, conditions, or nested functions.
- Reason:
  - React relies on call order to associate hook calls with internal state.

8.2 `useEffect` and Dependency Arrays

- `useEffect(effect)`:
  - Runs after every render by default.
- `useEffect(effect, [])`:
  - Runs only once after the initial render (componentDidMount).
- `useEffect(effect, [dep1, dep2])`:
  - Runs on initial render and whenever any dependency changes.
- Returning a cleanup function:
  - Runs before the effect re-runs and on unmount.
  - Use it to remove event listeners, cancel timers, etc.

8.3 `useContext`

- Context solves the problem of **prop drilling** (passing props through many layers).
- `useContext(SomeContext)` lets a component read values from the context provider without manual prop passing.

8.4 `useRef` vs `useState`

- `useState`:
  - Stores state.
  - Updating state triggers a re-render.
- `useRef`:
  - Holds a mutable `.current` value.
  - Updating `.current` does not trigger a re-render.
  - Common uses:
    - Storing DOM elements.
    - Storing timer IDs.
    - Storing previous values.

8.5 `useMemo` vs `useCallback`

- `useMemo`:
  - Memoizes a **computed value** based on dependencies.
  - Example: expensive calculations.
- `useCallback`:
  - Memoizes a **function** based on dependencies.
  - Helps prevent unnecessary re-renders when passing functions to child components.

8.6 State Lifting

- When two sibling components need shared data:
  - Move the shared state up to their closest common parent.
  - Parent holds state; passes it down as props.
  - Children communicate changes via callbacks.

8.7 `React.memo`

- `React.memo(Component)`:
  - Memoizes a functional component.
  - Skips re-render if props haven't changed (shallow comparison).
- Useful for performance optimizations with pure components.

8.8 Global State & Flux/Redux Concept

- Flux pattern:
  - **Actions**: describe what happened.
  - **Reducers**: pure functions that take (state, action) → new state.
  - **Store**: holds global state.
- In Redux-like systems:
  - State changes only via dispatched actions and reducers.
  - Reducers must be pure (no side effects).
  - Makes state changes predictable and easier to debug.

8.9 Advanced React Patterns

Higher-Order Components (HOCs):

- Function that takes a component and returns a new component.
- Used to share logic across components before hooks.
- Today, often replaced by custom hooks for logic sharing.

React Portals:

- Allow rendering children into a different part of the DOM.
- Used for modals, tooltips, dropdowns that need to escape parent stacking/overflow contexts.

Error Boundaries:

- Class components that catch errors in child components during rendering.
- Use `getDerivedStateFromError` and `componentDidCatch` to show fallback UI.

Code Splitting & Suspense:

- `React.lazy()` + `<Suspense>` allow lazy-loading components.
- Improves initial load time by only loading heavy components when needed.
- While loading, Suspense shows a fallback (spinner, skeleton, etc.).

---

9. Angular: Architecture & RxJS (High-Level)

(If you're focusing mostly on React, treat this as "bonus".)

9.1 Angular SPA Architecture

- Single index.html with a root element (e.g., `<app-root>`).
- Angular bootstraps the root module or standalone component into that element.
- Angular handles routing on the client side for SPA behavior.

9.2 Component, Module, and Decorators

- Component:
  - Class + template + styles + metadata (`@Component`).
- NgModule:
  - Groups components, directives, pipes, providers.
  - Declares, imports, and exports building blocks.
- Decorators (`@Input`, `@Output`):
  - `@Input`: parent → child data.
  - `@Output`: child → parent events via `EventEmitter`.

9.3 Data Binding, Directives, Pipes

- Data binding:
  - Interpolation: `{{ value }}`
  - Property binding: `[property]="value"`
  - Event binding: `(event)="handler($event)"`
  - Two-way: `[(ngModel)]="value"`
- Directives:
  - Structural (`*ngIf`, `*ngFor`): add/remove DOM elements.
  - Attribute (`ngClass`, `ngStyle`): change appearance/behavior.
- Pipes:
  - Transform values in templates (e.g., `{{ amount | currency }}`).

9.4 Services, Dependency Injection & RxJS

- DI:
  - Services injected into components.
  - `@Injectable({ providedIn: 'root' })` for app-wide singletons.
- Observables (RxJS):
  - Streams of values over time.
  - Used for HTTP, events, and more.
  - Advantages: multiple values, cancellation, operators like `map`, `switchMap`, etc.
- Subjects:
  - Both observer and observable, you can `next()` values.

    - `BehaviorSubject` holds current value and emits it to new subscribers.
- Async pipe:
    - `{{ data$ | async }}` automatically subscribes/unsubscribes to Observables in templates.

9.5 Angular Forms, Routing, and Optimization

- Template-driven vs Reactive forms:
    - Template-driven: logic mostly in templates, good for simple forms.
    - Reactive: form model in TypeScript, better for complex validation and testing.
- Routing:
    - Route guards (`CanActivate`) to protect routes.
    - Resolvers to load data before route activation.
    - Lazy loading modules using `loadChildren` for performance.
- Performance:
    - ChangeDetectionStrategy.OnPush for manual, optimized re-rendering.
    - ViewEncapsulation modes for CSS scoping.
    - Zone.js tracks async tasks to trigger change detection.

---

10. Design & Wireframing

10.1 Fidelity Levels

- Low-fidelity:
    - Rough sketches/wireframes.
    - Focus on layout, user flow, information structure.
    - Avoid detailed colors and typography at this stage.
- High-fidelity:
    - Detailed visuals, final colors, fonts, spacing.
    - Used later to communicate near-final design.
- Focusing on visual details too early:
    - Can distract from fixing major UX flaws.
    - May cause you to overinvest in designs that will change.

10.2 Purpose of Wireframes

- Primary goal:
    - Map out user flows, page structure, and information hierarchy.
    - Get alignment on what goes where and how users move through the product.

- Not meant to be:
    - Final visual design.
    - Pixel-perfect or branded.

Less Detail:

Web Dev Interview Cram Sheet

HTTP & Web Fundamentals

HTTP Lifecycle – URL → Page

- Parse URL → resolve **DNS** to IP.
- Open **TCP** (and **TLS** for HTTPS) connection.
- Send **HTTP request**: request line + headers + optional body.
- Server routes request → runs app logic → talks to DB/services.
- Server sends **HTTP response**: status line + headers + body (HTML/JSON/etc).
- Browser:
  - Parses HTML → builds **DOM**.
  - Loads CSS → **CSSOM**; runs JS.
  - Requests additional assets (CSS, JS, images, fonts).
  - Layout + paint → page rendered.

Statelessness & Cookies

- **Stateless**: each HTTP request is independent; server doesn't remember previous ones.
- No built-in "logged-in user" or session across requests.
- **Cookies**:
  - Small key–value pairs stored in browser.
  - Server sets via `Set-Cookie`; browser sends via `Cookie`.
  - Contain session IDs/tokens that server uses to restore user state.
  - Attributes: `Secure`, `HttpOnly`, `SameSite`, `Expires/Max-Age`.

GET vs POST

- **Data location**:
  - GET: data in URL (query string); visible in address bar/logs.
  - POST: data in body; not visible in URL.
- **Caching**:
  - GET: cacheable by default.
  - POST: typically not cached.
- **Semantics / idempotency**:

    - GET: safe + idempotent (no state changes).
    - POST: not necessarily idempotent (multiple POSTs can duplicate).

PUT vs PATCH

- **PUT**:
    - Replace entire resource.
    - Client sends full representation.
    - Typically idempotent.
- **PATCH**:
    - Partial update; send only changes.
    - Semantically "modify existing resource."

Content-Type vs Accept

- **Content-Type**:
    - Describes **body format being sent** (request or response).
    - e.g., `Content-Type: application/json`.
- **Accept**:
    - Client's preferred **response formats**.
    - e.g., `Accept: application/json, text/html`.

4xx vs 5xx + Examples

- **4xx**: client errors (problem with request).
    - 401 Unauthorized: not authenticated (missing/invalid token).
    - 403 Forbidden: authenticated but not allowed.
    - 404 Not Found: resource doesn't exist.
- **5xx**: server errors (server failed).
    - 500 Internal Server Error: unhandled exception/bug/DB crash.

3xx / 301 vs 302

- **3xx**: redirection; client should use another URL.
- **301 Moved Permanently**:
    - Permanent new location.
    - Browsers/SEO update links and cache redirect.
- **302 Found / Temporary Redirect**:

    - Temporary move.
    - Clients continue using original URL in future.

HTML Structure & Semantics

DOM Tree

- Browser parses HTML into **DOM**: a tree of nodes (elements, text, etc.) in memory.
- JavaScript manipulates the DOM via `document.*` APIs.
- DOM is the live structure the browser uses to render the page.

Block vs Inline

- **Block elements** (`<div>`, `<p>`, `<h1>`):
    - Start on a new line.
    - Fill available width.
    - Respect width/height and vertical margins.
- **Inline elements** (`<span>`, `<a>`, `<strong>`):
    - Flow inside text.
    - Only as wide as content.
    - Width/height behave differently.
- Semantically, inline **should not contain** block elements.

Semantic HTML / Accessibility / SEO

- Use `<header>`, `<nav>`, `<main>`, `<article>`, `<section>`, `<footer>` instead of only `<div>`.
- **Accessibility**:
    - Screen readers navigate by landmarks.
    - Clear structure for assistive tech.
- **SEO**:
    - Search engines better understand content and hierarchy.

Form Validation

- HTML5 attributes (`required`, `type="email"`, `pattern`) give **client-side** validation → better UX.

- Still need **server-side** validation:
  - Client checks can be bypassed.
  - Server must enforce security and data integrity.

`<label>` Tag

- Associate with `for` + `id`:
- `<label for="email">Email</label>`
- `<input id="email" type="email">`

  or wrap input in `<label>`.

- Benefits:
  - Clicking label focuses input.
  - Screen readers read label with input → better accessibility.

HTML5 Input Types & Mobile

- Types: `email`, `number`, `tel`, `date`, etc.
- Mobile browsers:
  - Show context-specific keyboards and pickers.
- Desktop:
  - Same semantics; may show spinners/date widgets.

CSS Styling & Layout

Box Model + `border-box`

- Box = **content + padding + border + margin**.
- Default (`content-box`):
  - Total width = width + padding + border (+ margins).
  - Example: width 200px + 20px padding each side → 240px total width (no border).
- `box-sizing: border-box`:
  - Declared width includes content + padding + border.
  - Same example → total stays 200px; content shrinks.

Specificity & Cascade

- Specificity order:
    1. Inline styles
    2. IDs (`#id`)
    3. Classes/attributes/pseudo-classes (`.class`, `[attr]`, `:hover`)
    4. Elements/pseudo-elements (`div`, `::before`)
- If ID, class, and element all target the same element: **ID wins**.
- Tie → later rule in CSS wins.

Flexbox vs Grid

- **Flexbox**:
    - One-dimensional (row OR column).
    - Best for navbars, toolbars, centering, simple rows/columns.
- **Grid**:
    - Two-dimensional (rows AND columns).
    - Best for page layouts, dashboards, galleries.

Positioning: `absolute` inside `relative`

- Parent: `position: relative`.
    - Stays in normal flow.
    - Becomes containing block.
- Child: `position: absolute`.
    - Removed from normal flow.
    - Positioned relative to **nearest non-static ancestor** (here, the relative parent) using `top`, `left`, etc.

Responsive Design & Media Queries

- Responsive site:
    - Fluid layouts (%, flex, grid).
    - Scales images and text.
    - Uses media queries for breakpoints.
- Media query:
- `@media (max-width: 768px) { ... }`
- **Mobile-first**:
    - Base styles for small screens.
    - Use `min-width` breakpoints to enhance for larger screens.

CSS Variables

- Declare:
- `:root { --main-color: #3498db; }`
- Use:
- `color: var(--main-color);`
- Benefits:
    - Centralized theming; easier global changes.
    - Cascade: variables can be overridden in specific containers (e.g., `.dark-theme { --main-color: black; }`).

Pseudo-classes vs Pseudo-elements

- **Pseudo-classes** (`:hover`, `:focus`, `:nth-child`):
    - Represent **state** of an element.
- **Pseudo-elements** (`::before`, `::after`, `::first-line`):
    - Represent **parts** of an element; can insert decorative content without extra markup.

JavaScript Core & Logic

var vs let vs const

- `var`:
    - Function-scoped.
    - Hoisted, initialized as `undefined`.
    - Not block-scoped.
- `let` and `const`:
    - Block-scoped.
    - Hoisted but not initialized (temporal dead zone).
    - `const` cannot be reassigned.

Hoisting

- Declarations are moved to top of scope at compile time.
- Function **declarations**:
    - Fully hoisted; can be called before they appear.

- `const`/`let` function expressions (e.g., arrow functions):
  - Variable hoisted but uninitialized → accessing before definition throws `ReferenceError`.

`this` Context

- Regular function:
  - `this` depends on call site:
    - `obj.method()` → `this = obj`.
    - `func()` → `this = undefined` in strict mode; `window` otherwise.
    - `.call/.apply/.bind` can override.
- Arrow function:
  - No own `this`; uses **lexical** `this` from surrounding scope.

== vs === (Type Coercion)

- `==`:
  - Coerces types before comparison.
  - e.g. `1 == '1'` → true, `0 == ''` → true.
- `===`:
  - No coercion; must be same type and value.
  - `1 === '1'` → false.
- Prefer `===` to avoid weird coercion bugs.

Spread vs Rest

- **Rest** (collect):
- `function myFunc(...args) { /* args is array */ }`
- **Spread** (expand):
- `const newArr = [...oldArr, 4, 5];`
- `const newObj = { ...oldObj, extra: true };`

Template Literals

- Use backticks:
- `const message = \`Hello, ${name}!\`;`
- Features:
  - Expression interpolation.
  - Multi-line strings without `\n`.

- Tagged templates (advanced).

Default Parameters

- Syntax:
- `function greet(name = 'Guest') { ... }`
- Passing `undefined` → uses default.
- Passing `null` → value is `null` (no default).

`"use strict"`

- Enables strict mode:
  - Disallows implicit global vars (e.g., `x = 10`).
  - Throws more errors instead of silently failing.
  - Changes some `this` behavior (`this` is `undefined` in plain functions).

JS DOM & Async

Bubbling vs Capturing

- **Capturing**: event goes from root → target.
- **Bubbling**: event goes from target → root (default phase for listeners).
- `event.stopPropagation()`:
  - Stops further propagation (in capturing or bubbling).

Event Delegation

- Add one listener on a parent (e.g., `<ul>`), use `event.target` to detect which child was clicked.
- Benefits:
  - Fewer event listeners → better performance.
  - Works automatically for dynamically added children.

Promises vs Async/Await

- Promises:
  - Use `.then()` / `.catch()`.
- Async/await:
  - Same underlying Promises; more readable:
  - `try {`
  - `  const res = await fetch(url);`
  - `} catch (err) { ... }`
  - Error handling with `try/catch` → resembles synchronous code.

Fetch API Errors

- `fetch()`:
  - **Rejects** on network errors only.
  - 404/500 still **resolve**; you must check `response.ok`/`status`.
- Typical pattern:
- `const res = await fetch(url);`
- `if (!res.ok) throw new Error(res.statusText);`

JSON Handling

- Valid JSON types:
  - Object, array, number, string, boolean, `null`.
- Cannot represent:
  - Functions, `undefined`, Symbols, class methods.
- Convert:
  - `JSON.stringify(obj)` → string.
  - `JSON.parse(str)` or `response.json()` → JS value.

TypeScript

TS vs JS

- TypeScript is a typed superset of JS.
- TS compiler:
  - Type-checks your code.
  - Compiles TS → plain JS.
- Browsers run **JavaScript**, not TypeScript.

`any` VS `unknown`

- `any`:
  - Turns off type checking; you can do anything with it.
- `unknown`:
  - Must be **narrowed** before use (`typeof`, `instanceof`, type guards).
  - Safer because you can't accidentally misuse it.

Interfaces vs Type Aliases

- Both can define object shapes.
- Prefer **interface** when:
  - You want declaration merging.
  - You expect others to extend it.
- Use `type` when:
  - You need unions, intersections, or more complex type compositions.

Union Types

- Type can be one of several types:
- `let id: string | number;`
- Must narrow before using:
- `if (typeof id === 'string') { ... }`

Tuples

- Fixed length + fixed types per position:
- `let pair: [string, number];`
- Different from `(string | number)[]`, which allows any length and order.

Type Guards

- Function returning `arg is Type`:
- `function isUser(x: any): x is User {`
- `  return x && typeof x.name === 'string';`
- `}`
- Inside `if (isUser(obj)) {}`, TS treats `obj` as `User`.

Generics & Constraints

- Array of strings (generic syntax):
- `const arr: Array<string> = [];`
- Generic function:
- `function identity<T>(value: T): T { return value; }`
- Constraint:
- `function logLength<T extends { length: number }>(val: T) {`
- `  console.log(val.length);`
- `}`

`keyof`, Mapped Types, Utilities

- `keyof`:
- `type Keys = keyof User; // 'id' | 'name' | ...`
- Mapped types:
- `type ReadonlyUser = { [K in keyof User]: Readonly<User[K]> };`
- Utilities:
  - `Partial<T>`: all props optional (good for updates).
  - `Pick<T, K>`: only props K.
  - `Omit<T, K>`: all except K.

`tsconfig target`

- Controls JS version output (ES5, ES2015, ES2020, etc.).
- Affects:
  - Syntax transforms.
  - Which features are downleveled for older environments.

React: Core Concepts & Components

Virtual DOM & Diffing

- Virtual DOM:
  - In-memory representation of UI.
- On state change:
  - React builds new VDOM.
  - Diffs vs previous VDOM.
  - Applies minimal changes to real DOM (reconciliation).
- Fewer direct DOM ops → better performance.

JSX

- JSX is syntax sugar for `React.createElement`.
- Not understood by browsers; compiled (e.g., via Babel) to JS function calls.

Props vs State

- **Props**:
  - Inputs from parent.
  - Read-only in child.
- **State**:
  - Internal data managed by component (`useState`).
  - Changes trigger re-render.

Unidirectional Data Flow

- Data flows **down** via props.
- Children notify parents via callbacks (functions passed as props).
- Makes state flow predictable and easier to debug.

Controlled vs Uncontrolled Components

- Controlled:
  - Input value controlled by React state (`value`, `onChange`).
  - Source of truth is state.
- Uncontrolled:
  - DOM manages the input value; access via refs.
- Controlled is generally recommended for non-trivial forms.

Keys in Lists

- `key` identifies list items across renders.
- Good key: stable, unique ID from data.
- Using index:
  - Breaks when reordering/inserting items.
  - Can cause incorrect re-use of DOM nodes (weird bugs).

Synthetic Events

- React wraps native events in SyntheticEvent.
- Provides consistent cross-browser behavior.
- Historically used event pooling; now mostly just a standardized wrapper.

React: Hooks & Lifecycle

Rules of Hooks

- Only call hooks:
  - In React function components or custom hooks.
  - At the top level (not inside loops/conditions).
- React relies on order of hook calls to associate state with them.

`useEffect` Dependency Array

- No array: run after every render.
- `[]`: run once after first render (mount).
- `[dep1, dep2]`: run when any dependency changes.
- Return cleanup function to remove listeners/subscriptions before re-run/unmount.

`useContext`

- Avoids prop drilling.
- Provides global-ish value (theme, user, locale).
- `useContext(SomeContext)` reads nearest provider's value.

`useRef` VS `useState`

- `useState`:
  - Changes trigger re-render.
- `useRef`:
  - Stores mutable `.current` value without re-rendering.
  - Use for DOM refs, timers, "instance" variables.

`useMemo` VS `useCallback`

- `useMemo`:
  - Memoizes a **value**.
  - `const result = useMemo(() => compute(...), [deps]);`
- `useCallback`:
  - Memoizes a **function**.
  - `const handler = useCallback(() => {...}, [deps]);`
- Both used for performance optimization (prevent unnecessary recalculations/re-renders).

React: State Management & Performance

State Lifting

- When siblings share data:
  - Move state up to closest common parent.
  - Parent holds state and updater.
  - Pass data + callbacks down via props.

`React.memo`

- Wraps a component to skip re-render if props are shallowly equal.
- Good for pure components with heavy render logic.

Flux / Redux Concepts

- **Action**: plain object describing what happened.
- **Reducer**: pure function `(state, action) => newState`.
- **Store**: holds app state, dispatches actions to reducers.
- Reducers must be pure → easier to test and reason about.

React: Advanced Patterns & Scenarios

HOCs vs Custom Hooks

- HOC:

    - Function that takes a component and returns a new component with added behavior.
- Custom hook:
    - Function that encapsulates reusable logic using hooks.
- Today: prefer custom hooks for logic reuse; simpler component trees.

React Portals

- Render children into a DOM node outside the parent hierarchy.
- Useful for modals, tooltips that must escape parent `overflow`/stacking contexts.

Error Boundaries

- Class components that catch errors in child_render/lifecycle.
- Use `getDerivedStateFromError` + `componentDidCatch`.
- Show fallback UI instead of crashing the whole app.

Code Splitting & Suspense

- `React.lazy(() => import('./Comp'))` + `<Suspense fallback={...}>`.
- Lazy-loads component code on demand → smaller initial bundle.
- While loading, Suspense shows the fallback (e.g., spinner).

Performance Optimization Scenario (10,000 items)

- Debounce input changes.
- Memoize filtered results with `useMemo`.
- Virtualize list (render only visible items) with libraries like `react-window`.
- Use `React.memo` on list items where appropriate.

Testing Scenario (API call on mount)

- Mock API calls:
    - Tests should be fast, deterministic, and not hit real servers.
- Test flow:

- Assert initial “loading” state.
- Resolve mock → assert final data rendered and loading removed.

Angular (High-Level Only – If Needed)

(If you don’t need Angular, you can stop here.)

Core Ideas to Remember

- **SPA Architecture**: Angular bootstraps into `<app-root>` in `index.html`.
- **Components vs Modules**: Components declared in NgModules (unless standalone); modules group components/services/pipes.
- **Bindings**:
  - `{{ }}` – interpolation.
  - `[prop]` – property binding.
  - `(event)` – event binding.
  - `[(ngModel)]` – two-way binding.
- **Directives**:
  - Structural (`*ngIf`, `*ngFor`): modify DOM structure.
  - Attribute (`ngClass`, `ngStyle`): modify appearance/behavior.
- **Pipes**: transform values in templates (`| date`, `| currency`).
- **DI & Services**:
  - `@Injectable({ providedIn: 'root' })` for app-wide singleton.
- **Observables**:
  - Streams with RxJS; `subscribe()` needed to execute.
  - `async` pipe in templates auto-subscribes/unsubscribes.
- **Route Guards**:
  - `CanActivate` to protect routes (e.g., `/dashboard`).
- **Change Detection**:
  - `OnPush` strategy for performance; triggers on input reference changes/events.
- **Shared Service for Widget Communication**:
  - Use Subject/BehaviorSubject in a shared service for broadcasting messages to a notification bar.

Design & Wireframing

Fidelity Levels

- **Low-fidelity**:
  - Sketches/wireframes.
  - Focus on layout, flow, and information hierarchy.

- **High-fidelity**:
  - Colors, typography, spacing, near-final look.
- Focusing on visuals too early can hide UX issues and slow iteration.

Wireframe Purpose

- Primary goal:
  - Map user flow and information architecture.
  - Decide what goes where and how users move through the app.
- Not meant to show final polished visuals.

Even Less Details:

Web Dev Interview – One-Page Cram Sheet

HTTP & Web Fundamentals

- **Lifecycle (URL → Page)**: URL parsed → DNS to IP → TCP/TLS → send request (line + headers + optional body) → server handles (routing + DB) → response (status + headers + body) → browser builds DOM + CSSOM → layout/paint → run JS → load extra assets.
- **Statelessness & Cookies**: HTTP doesn't remember previous requests. Cookies (`Set-Cookie` / `Cookie`) store session IDs/tokens so the server can associate requests with user state.
- **GET vs POST**:
  - GET: data in URL, cacheable, safe + idempotent.
  - POST: data in body, not cacheable by default, not idempotent.
- **PUT vs PATCH**: PUT = full replace (send whole resource, idempotent). PATCH = partial update (send only changes).
- **Content-Type vs Accept**: `Content-Type` = format of what is **sent**; `Accept` = what client wants in **response**.
- **4xx vs 5xx**: 4xx = client error (e.g., 401 unauthenticated, 403 unauthorized, 404 missing). 5xx = server error (500 bug/exception).
- **3xx / 301 vs 302**: 3xx = redirect. 301 = permanent; clients update URL. 302 = temporary; keep original URL.

HTML & Semantics

- **DOM Tree**: Browser parses HTML into an in-memory **DOM** tree; JS manipulates this tree to change the UI.
- **Block vs Inline**: Block (div, p) → new line, full width; inline (span, a) → flow in text. Inline shouldn't contain block.
- **Semantic Tags**: `<nav>`, `<main>`, `<article>`, `<footer>` improve a11y (screen readers) + SEO + code clarity.
- **Form Validation**: Client-side (HTML5/JS) for UX; **must** still validate on server for security and integrity.
- **`<label>`**: `for="id"` or wrap input. Improves focus on click + screen reader announcements.
- **Input Types**: `email`, `number`, `date`, etc. give better native controls and mobile keyboards.

CSS & Layout

- **Box Model**: Total width = content + padding + border (+ margin). `box-sizing: border-box` makes declared width include padding + border.
- **Specificity**: inline > `#id` > `.class`/`:hover` > `element`. Tie → last rule wins.
- **Flexbox vs Grid**: Flexbox = 1D (row OR column). Grid = 2D (rows AND columns).
- **Positioning**: Parent `position: relative`; child `position: absolute` → positioned relative to that parent, removed from normal flow.
- **Responsive + Media Queries**: Fluid units + `@media (max-width: ...)`. Mobile-first = base for small screens, enhance with `min-width`.
- **CSS Variables**: `--var` + `var(--var)` enable theming and easy overrides via cascade.
- **Pseudo-class vs Pseudo-element**: `:hover`/`:focus` = element state; `::before`/`::after` = virtual sub-elements.

JavaScript Core

- **var / let / const**: `var` = function-scoped, hoisted. `let`/`const` = block-scoped, TDZ. `const` not reassignable.
- **Hoisting**: Function declarations callable before definition; `const`/`let` function expressions are NOT.
- **`this`**: Regular function `this` depends on call site; arrow function `this` is lexical (from surrounding scope).
- **== vs ===**: `==` coerces; `===` doesn't. Prefer `===` to avoid surprise.
- **Spread vs Rest**: Rest collects (`function f(...args)`), spread expands (`[...arr]`, `{...obj}`).
- **Template Literals**: Backticks → `${expr}` interpolation + multi-line strings.
- **Default Params**: `function f(x = 5) {}`; `undefined` triggers default, `null` does not.
- **"use strict"**: Stricter semantics (no implicit globals, safer `this`, more errors instead of silent failures).

DOM & Async

- **Bubbling vs Capturing**: Capturing: root→target; bubbling: target→root. Default listeners = bubbling. `stopPropagation()` stops further travel.
- **Event Delegation**: Attach one handler on parent; check `event.target` for child. Fewer listeners + works with dynamic elements.
- **Promises vs Async/Await**: Async/await is syntactic sugar over promises; `try/catch` makes async error handling look synchronous.
- **Fetch + Errors**: `fetch` rejects only on network error; 404/500 still resolve. Must check `response.ok`/`status`.
- **JSON**: Valid JSON: objects, arrays, numbers, strings, booleans, `null`. No functions/`undefined`/symbols.

TypeScript

- **TS vs JS**: TS adds static types; compiler type-checks then emits plain JS. Browsers run JS only.
- **`any` vs `unknown`**: `any` = no safety; `unknown` requires type checks before use, so safer.
- **Interfaces vs Types**: Both define shapes; interfaces support extension/merging; types handle unions/intersections.
- **Union Types**: `string | number`; must narrow with checks before using.
- **Tuples**: `[string, number]` = fixed length and positions; different from `(string | number)[]`.
- **Type Guards**: Functions returning `arg is MyType` let TS narrow unions inside conditionals.
- **Generics + Constraints**: `Array<string>`, `function f<T>(x: T)`. Use `T extends { length: number }` when you need `.length`.
- **`keyof` + Utilities**: `keyof T` = union of property names. `Partial<T>` (all optional), `Pick<T,K>` (subset), `Omit<T,K>` (minus keys).
- **`tsconfig target`**: Controls JS version output (ES5/ES2015/ES2022), i.e., what syntax/features TS downlevels.

React Core

- **Virtual DOM**: React builds VDOM → diffs with previous → minimal real DOM updates = better performance.
- **JSX**: Compiles to `React.createElement(...)`; build step converts JSX to JS.
- **Props vs State**: Props = read-only inputs from parent; state = internal, updated via `setState`/`useState`.
- **Unidirectional Data Flow**: Data flows parent→child via props; children report up via callbacks.
- **Controlled vs Uncontrolled**: Controlled inputs use React state as source of truth. Uncontrolled use DOM value + refs. Controlled usually preferred.
- **Keys**: Unique, stable `key` for list items so React can track moved/changed items; avoid array index.

React Hooks & Performance

- **Rules of Hooks**: Call only at top level of React function components/custom hooks (no loops/ifs), and always in same order.
- **`useEffect`**: No deps = every render; `[]` = on mount; `[deps]` = when deps change. Return cleanup for unsubscribes/listeners.
- **`useContext`**: Fixes prop drilling by reading provider value directly.

- **`useRef` vs `useState`**: `useState` triggers re-render; `useRef` stores mutable value without re-render (DOM nodes, timers, previous values).
- **`useMemo` vs `useCallback`**: `useMemo` memoizes values; `useCallback` memoizes function references.
- **State Lifting**: Move shared state to common parent; pass data + callbacks down to siblings.
- **`React.memo`**: Skips re-render if props shallowly equal (for pure components).
- **Flux/Redux Idea**: Action (what happened) → reducer (pure function) → new state in store; predictable state changes.
- **Performance Scenario**: Big list + search → debounce input, `useMemo` filtered results, virtualize list, `React.memo` row components.
- **Testing Scenario**: Mock API calls (no real network), test loading state first, then success/error states after mocked resolution.

Design & Wireframing

- **Fidelity**: Low-fi = layout & flow; hi-fi = colors, fonts, details. Don't obsess over visuals too early.
- **Wireframe Purpose**: Map user flow and information architecture—not final visual design.