110 JavaScript Developer Standards — Practical, Secure, and Maintainable (Focused on JavaScript)

Generated: 2025-10-10

How to use this document

This PDF lists 110 practical standards for JavaScript developers. Each standard is concise and actionable. Use them in lint rules, code reviews, CI checks, and internal guidelines. Sections include naming, packages & dependencies, date/time, security violations, code quality, testing, architecture, and domain-specific naming guidance.

General coding standards

1. Keep functions small (single responsibility). Aim for ≤ 50 lines. 2. Prefer pure functions where possible (no side effects). 3. Use immutable data patterns for state updates (avoid mutating inputs). 4. Use explicit return values — avoid implicit undefined. 5. Prefer expressions over statements where it improves clarity. 6. Use ES modules (import/export) consistently. 7. Avoid over-abstraction — YAGNI (you aren't gonna need it). 8. Document non-obvious decisions with a short comment above the code. 9. Use feature flags or config for behavior toggles, not ad-hoc booleans. 10. Make APIs predictable: validate inputs and provide clear errors.

Naming conventions

11. Use camelCase for variables and functions (e.g., userCount, fetchData). 12. Use PascalCase for class/constructor and React components (e.g., UserService, UserCard). 13. Prefix boolean-returning functions with `is`/`has`/`can` (e.g., isEmpty, hasPermission). 14. Use `get`/`set` for accessors (getUser, setConfig). 15. Use verbs for functions and actions (createUser, fetchOrders). 16. Use singular nouns for single entities and plural for collections (user, users). 17. Avoid abbreviations unless universally known (use totalAmount, not totAmt). 18. Use domain-specific names (libraryBook vs genericItem) to convey intent. 19. For constants use UPPER_SNAKE_CASE (MAX_RETRIES). Prefer `const` declarations. 20. Name error variables `err`/`error` consistently in callbacks/promises.

Files & folders

21. Keep one top-level responsibility per file (module-per-concept). 22. Name files after their default export (UserService.js -> export default UserService). 23. Use index.js only for folder re-exports, avoid large barrel files that hide dependencies. 24. Group related components, hooks, and tests in the same folder. 25. Keep public API modules at a clear entry point (src/index.js).

Formatting & style

26. Use consistent indentation (2 or 4 spaces across project) configured via Prettier. 27. Enforce styling with Prettier + ESLint in CI. 28. Keep line length reasonable (80–120 chars) for readability. 29. Use trailing commas only where your formatter enforces them for cleaner diffs. 30. Prefer template literals over string concatenation for multi-part strings.

Dependencies & packages

31. Only import what you need — avoid `import * as util from 'util'` when you need one function. 32. Prefer named imports to keep tree-shaking effective. 33. Avoid `npm update --force` blind upgrades; check changelogs. 34. Remove unused packages — run `depcheck` or `eslint-plugin-unused-imports`. 35. Pin direct dependencies in package.json if reproducibility matters (use lockfile for exact versions). 36. Use devDependencies for tooling, dependencies for runtime. 37. Prefer small, focused libraries over large frameworks for single features. 38. Avoid publishing secrets or credentials in package.json or `.env`. 39. Do not depend on packages without recent maintenance for critical code paths. 40. Use package audit tools in CI (npm audit / Snyk) and treat high severity findings as blockers.

Date & time formats

41. Use ISO 8601 for string timestamp interchange (e.g., 2025-10-10T14:30:00Z). 42. Standardize on UTC for storage, convert to local timezone only at presentation layer. 43. Use libraries intentionally: prefer native `Date` + Intl for simple cases; use `luxon` or `date-fns` for complex manipulation. Avoid moment.js for new projects. 44. Normalize incoming date strings on API boundaries (validate format). 45. When serializing for APIs, use full timestamps (YYYY-MM-DDTHH:mm:ss.sssZ) rather than date-only when time matters. 46. For durations use ISO 8601 durations (e.g., P3DT4H) or integer milliseconds with units documented. 47. Document date formats in API contracts (OpenAPI schema example). 48. Agree on locale formats for UIs and do not mix (e.g., en-GB vs en-US). 49. Store timezone separately if necessary to reconstruct local times. 50. Avoid floating timezone arithmetic; use server-side libraries to shift times reliably.

Security — input & output handling

51. Validate inputs: use schema validators (ajv, zod) on boundaries. 52. Sanitize outputs that are rendered as HTML (prevent XSS). 53. Treat all external inputs as untrusted — never assume well-formed. 54. Use Content Security Policy (CSP) headers to limit script sources. 55. Escape values when inserting into HTML templates or using innerHTML. 56. Prefer safer DOM APIs (textContent) instead of innerHTML unless sanitized. 57. Use HTTPS everywhere — enable HSTS. 58. Limit data sent to clients — follow least privilege for API responses. 59. Use same-site cookies and secure flags for session cookies. 60. Rotate credentials and secrets periodically; store them in a vault (do not check into VCS).

Security — common violations (to detect and avoid)

61. Hard-coded credentials in source. 62. Unsanitized user input rendered in the DOM (XSS). 63. Improper use of eval(), new Function(), or setTimeout with code strings. 64. Insecure deserialization of untrusted data. 65. Excessive privileges in CORS allowing arbitrary origins. 66. Using `innerHTML` with untrusted content. 67. Insufficient rate limiting on public endpoints (brute force). 68. Relying solely on client-side validation for security. 69. Exposing stack traces or debug logs to production responses. 70. Including third-party scripts with unknown integrity without SRI.

Authentication & Authorization

71. Prefer proven auth solutions (OAuth2/OpenID Connect) over homegrown. 72. Use short-lived access tokens and refresh tokens with rotation. 73. Use scopes/roles for fine-grained authorization, not boolean flags sprinkled in code. 74. Enforce server-side authorization checks for every protected action. 75. Fail closed: default to deny unless explicitly allowed. 76. Validate JWT signatures and claims (iss, aud, exp). 77. Revoke sessions/refresh tokens on major account changes (password reset). 78. Log suspicious auth events and alert on abnormal patterns. 79. Avoid storing sensitive info in localStorage if XSS risk is present; consider httpOnly cookies. 80. Require multi-factor authentication for high-privilege actions where practical.

Error handling & logging

81. Use structured logs (JSON) for easier ingestion and search. 82. Do not leak PII or secrets into logs. Mask or redact sensitive fields. 83. Use error types and codes for machine-readable handling (e.g., ERR_... codes). 84. Provide user-friendly error messages; keep internal details in logs only. 85. Normalize errors at boundaries and map to HTTP/S status codes consistently. 86. Implement retry with exponential backoff and jitter for transient failures. 87. Ensure logs include correlation IDs to trace requests across services. 88. Monitor log volumes and set alerting for error spikes. 89. Use Sentry or similar for crash/error aggregation and triage. 90. Include breadcrumbs and context in error reports (userId, requestId).

Code quality & reviews

91. Use static analysis (ESLint) with strict, project-specific rules. 92. Enforce type safety: prefer TypeScript or JSDoc-annotated JS in large codebases. 93. Require code review for all non-trivial changes; use PR templates to surface security/impact questions. 94. Automate style and lint checks in CI and block merges on failures. 95. Keep PRs small and focused (aim ≤ 300 lines changed). 96. Use mutation or property-based testing for critical logic where helpful. 97. Use feature branches and CI gating; avoid long-lived forks. 98. Keep high-level architecture diagrams up-to-date in repo docs. 99. Use dependency graph tooling to understand the risk surface of third-party packages. 100. Introduce coding standards incrementally with automated autofixes where possible.

Testing & observability

101. Maintain a test pyramid: many unit tests, fewer integration tests, minimal E2E tests for critical flows. 102. Use test fixtures and seeded data for deterministic tests. 103. Run tests in CI on PRs and on a schedule for flaky detection. 104. Use contract tests for services with strict API agreements. 105. Use chaos or fault-injection testing selectively for resilience validation. 106. Add metrics and tracing for long-running flows (use OpenTelemetry or equivalent). 107. Set SLOs and monitor observable ratios (error rate, latency, saturation). 108. Ensure logs, metrics, and traces are correlated by request id. 109. Automate security scanning in CI: SAST, dependency checks, secret scanning. 110. Keep a documented rollback/incident runbook and practice it with game days.