Coding Standards

Coding Standards

> **Project:** {{PROJECT_NAME}} **Version:** {{VERSION}} **Date:** {{DATE}}
> **Author:** {{AUTHOR}} **Status:** Draft | In Review | Approved **Reviewers:** {{REVIEWERS}}

Document History

Version	Date	Author	Changes
0.1	{{DATE}}	{{AUTHOR}}	Initial draft

Purpose

These standards apply to all code in **{{PROJECT_NAME}}**. When automated tools enforce a rule, the tool wins. When in doubt, optimize for readability — the next person reading your code may be you in six months.

Non-negotiable: All new code must pass automated linting and formatting checks before merge. No exceptions.

---

1. Language-Specific Conventions

1.1 Naming Conventions

JavaScript / TypeScript:

Type	Convention	Example

Variables	`camelCase`	`userName`, `isLoading`
Functions	`camelCase`	`getUserById()`, `validateEmail()`
Classes	`PascalCase`	`UserService`, `PaymentProcessor`
Interfaces	`PascalCase` (no `I` prefix)	`User`, `PaymentResult`
Types	`PascalCase`	`UserId`, `ApiResponse<T>`
Enums	`PascalCase`	`UserRole`, `PaymentStatus`
Constants	`UPPER_SNAKE_CASE`	`MAX_RETRY_COUNT`, `API_BASE_URL`
Files — components	`PascalCase.tsx`	`UserProfile.tsx`
Files — utilities	`camelCase.ts`	`formatDate.ts`, `useAuth.ts`

Python:

Type	Convention	Example
Variables / functions	`snake_case`	`user_name`, `get_user_by_id()`
Classes	`PascalCase`	`UserService`
Constants	`UPPER_SNAKE_CASE`	`MAX_RETRY_COUNT`
Modules / files	`snake_case`	`user_service.py`
Private	`_single_leading_underscore`	`_internal_helper()`

1.2 File Organization

```
src/
├── {{MODULE_1}}/
│   ├── index.ts           # Public exports only
│   ├── {{MODULE_1}}.service.ts   # Business logic
│   ├── {{MODULE_1}}.controller.ts # HTTP handlers
│   ├── {{MODULE_1}}.types.ts     # Types/interfaces
│   ├── {{MODULE_1}}.test.ts      # Unit tests
│   └── {{MODULE_1}}.integration.test.ts
├── shared/                # Shared utilities, no business logic
│   ├── errors/
│   ├── utils/
│   └── types/
└── config/                # Configuration loading
```

Rules:

- One class/component per file
- File name matches the primary export name
- Test files co-located with source files (not in a separate `/test` directory)
- `index.ts` files only for re-exporting — no logic

1.3 Import Ordering

```
// 1. Node built-ins
import { readFileSync } from 'fs';
import path from 'path';

// 2. External dependencies (node_modules)
import express from 'express';
import { z } from 'zod';

// 3. Internal — absolute paths
import { UserService } from '@/services/user.service';
import { logger } from '@/shared/logger';

// 4. Internal — relative paths
import { validateRequest } from '../middleware/validate';
import type { CreateUserDto } from './user.types';
```

Enforced by: `eslint-plugin-import` / `isort` (Python)

1.4 Error Handling Patterns

Pattern: Return typed errors, never throw in business logic unless truly exceptional.

```
// PREFERRED — typed result pattern
type Result<T, E = Error> = { ok: true; value: T } | { ok: false; error: E };

async function createUser(dto: CreateUserDto): Promise<Result<User, ValidationError |
DatabaseError>> {
  const validation = validateUser(dto);
  if (!validation.ok) return { ok: false, error: new ValidationError(validation.message) };
  // ...
}

// ACCEPTABLE — throw only at application boundaries (HTTP handlers)
```

```
// FORBIDDEN — swallow errors silently
try {
  doSomething();
} catch (err) {
  // Never write this without handling the error
}
```

Async/await: Always use `try/catch` in async functions at the controller layer. **Promise chaining:** Avoid — prefer async/await. **Logging:** Log errors with `error` level at the boundary. Do NOT re-log in nested functions.

---

2. Code Formatting

2.1 Formatter

Language	Tool	Config File	Enforced In
TypeScript / JavaScript	{{FORMATTER}}	`{{CONFIG_FILE}}`	CI + pre-commit
Python	Black + isort	`pyproject.toml`	CI + pre-commit
JSON / YAML	Prettier	`.prettierrc`	CI + pre-commit

Key formatting rules:

- Indentation: {{INDENT}}
- Max line length: {{LINE_LENGTH}} characters
- Semicolons: {{SEMICOLONS}}
- Trailing commas: {{TRAILING_COMMAS}}
- Single quotes: {{QUOTES}}

Auto-format on save: Recommended — configure your IDE with the project's formatter settings.

2.2 Linter

Language	Tool	Config	Rules Severity
TypeScript	{{LINTER}}	`{{CONFIG_FILE}}`	Error = blocks CI, Warn = review
Python	Ruff / Flake8	`pyproject.toml`	Error = blocks CI

Linting rules that are errors (block CI):

- `no-unused-variables`
- `no-explicit-any` (TypeScript)
- `no-console` in production code
- Import ordering violations
- Type safety violations

Disable linting inline (sparingly):

```
// eslint-disable-next-line no-console -- debugging production issue, remove before merge
console.log('Debug:', data);
```

Every inline disable must have a comment explaining why.

2.3 Editor Config

```
# .editorconfig (root of repo)
root = true

[*]
indent_style = {{INDENT_STYLE}}
indent_size = {{INDENT_SIZE}}
end_of_line = lf
charset = utf-8
trim_trailing_whitespace = true
insert_final_newline = true
```

---

3. Git Conventions

3.1 Commit Message Format

Standard: Conventional Commits

Format:

```
<type>(<scope>): <description>

[optional body]
```

```
[optional footer(s)]
```

Types:

Type	When to Use
`feat`	New feature
`fix`	Bug fix
`docs`	Documentation only
`style`	Formatting changes (no logic change)
`refactor`	Code change that neither fixes a bug nor adds a feature
`test`	Adding or updating tests
`chore`	Build system, dependency updates, CI changes
`perf`	Performance improvements
`ci`	CI/CD configuration changes

Examples:

```
feat(auth): add magic link login flow
fix(cart): prevent double-charge on network retry
docs(api): update user endpoint examples
chore: upgrade typescript to 5.3
```

Breaking changes:

```
feat(api)!: rename user endpoint from /users to /accounts

BREAKING CHANGE: All clients must update API calls from /api/users to /api/accounts
```

Enforced by: commitlint + husky pre-commit hook

3.2 Branch Naming Convention

Type	Pattern	Example
Feature	`feature/{{TICKET}}-short-description`	`feature/AUTH-123-magic-link`
Bug fix	`fix/{{TICKET}}-short-description`	`fix/CART-456-double-charge`
Hotfix	`hotfix/{{TICKET}}-short-description`	`hotfix/SEC-789-xss-fix`
Release	`release/v{{VERSION}}`	`release/v2.4.0`
Chore	`chore/{{TICKET}}-short-description`	`chore/DEP-012-upgrade-react`

Rules:

- Lowercase only
- Hyphens, not underscores
- Include ticket number
- Keep description short (< 5 words)

3.3 PR Title & Description Format

Title format: Same as commit message format: `type(scope): description`

PR template: `.github/pull_request_template.md` — auto-loaded on PR creation

Required PR description sections:

1. **What**: What does this PR do? (1-3 bullet points)
2. **Why**: Why was this change made? (context, ticket link)
3. **How to test**: How should the reviewer verify it works?
4. **Screenshots**: Required for UI changes
5. **Checklist**: Pre-merge checklist (auto from template)

3.4 PR Size Guidelines

Size	Lines Changed	Status	Action
Small	< 200	Ideal	Review same day
Medium	200-500	Acceptable	Review within 24h
Large	500-1000	Needs justification	Split if possible
Extra Large	> 1000	Exceptional only	Must be pre-approved

Tips for keeping PRs small:

- One PR per feature/fix
- Separate refactoring from feature work
- Use feature flags to deploy incomplete features

---

4. Code Review Guidelines

4.1 What to Look For

Reviewers should check:

- [ ] Logic is correct and handles edge cases
- [ ] Tests cover the change (and edge cases)
- [ ] No obvious security issues (input validation, auth checks, SQL injection)
- [ ] Error handling is appropriate
- [ ] Performance: no N+1 queries, no unbounded operations
- [ ] Code follows naming and organization standards
- [ ] Documentation updated if behavior changed

Reviewers should NOT block on:

- Personal style preferences that aren't in the standards
- Minor refactors not related to the PR's scope
- Hypothetical future requirements

4.2 Review Turnaround Expectations

PR Type	First review	Re-review after changes
Critical / Hotfix	Within {{HOTFIX_SLA}} hours	Within {{HOTFIX_RESLA}} hours
Standard	Within {{STANDARD_SLA}} business hours	Within {{STANDARD_RESLA}} hours

4.3 Approval Requirements

Branch	Required Approvals	Notes
`main` / `develop`	{{MAIN_APPROVALS}} (including CODEOWNER)	
Feature branches	{{FEATURE_APPROVALS}}	
Hotfix	{{HOTFIX_APPROVALS}} (emergency: 1)	Post-merge full review

4.4 Constructive Review Feedback

Use these prefixes to set expectations:

- `nit:` — Minor issue, not a blocker: `nit: prefer const here`
- `question:` — Need clarification, not a change request: `question: why did you choose X over Y?`
- `suggestion:` — Improvement idea, not required: `suggestion: this could be simplified with X`
- `blocker:` — Must be fixed before merge: `blocker: this allows SQL injection`

Tone guidelines:

- Comment on code, not the author
- Explain the why: "This can cause N+1 queries, which will slow down under load" > "Wrong"
- Acknowledge good code: "Nice approach here — much cleaner than what we had"
- Be kind, be direct, be specific

---

5. Testing Standards

5.1 Test Naming Conventions

```
// Format: it('should [expected behavior] when [condition]')
it('should return 404 when user does not exist', async () => { ... });
it('should hash the password before storing', async () => { ... });
it('should throw ValidationError when email is invalid', async () => { ... });

// Describe blocks for grouping
describe('UserService', () => {
  describe('createUser', () => {
    it('should create user with valid input', async () => { ... });
    it('should throw when email already exists', async () => { ... });
  });
});
```

5.2 Test File Organization

```
src/
├── users/
│   ├── user.service.ts
│   ├── user.service.test.ts      # Unit tests
│   └── user.service.integration.test.ts  # Integration tests
```

5.3 Mocking Guidelines

```
// PREFERRED: Mock at the module level
jest.mock('../external/email-service');
```

```
// PREFERRED: Use test doubles that match the interface
const mockEmailService: EmailService = {
  send: jest.fn().mockResolvedValue({ id: 'msg-123' }),
};

// AVOID: Over-mocking internals
// AVOID: Tests that test mocks, not behavior
```

5.4 Coverage Requirements

Layer	Lines	Branches	Notes
Business logic	≥ {{BIZ_COV}}%	≥ {{BIZ_BRANCH}}%	Strictly enforced
Controllers / handlers	≥ {{CTRL_COV}}%	≥ {{CTRL_BRANCH}}%	
Utilities	≥ {{UTIL_COV}}%		
Overall minimum	≥ {{MIN_COV}}%		CI gate

---

6. Documentation Standards

6.1 When to Add Comments

```
// GOOD — explains WHY (not obvious from code)
// Retry up to 3 times because the payment API has transient failures on high load
for (let i = 0; i < 3; i++) { ... }

// BAD — restates WHAT (obvious from code)
// Loop 3 times
for (let i = 0; i < 3; i++) { ... }

// GOOD — documents non-obvious behavior
// Note: This function mutates the input array for performance. Callers must not
// pass arrays they intend to use afterwards.
function sortInPlace(arr: number[]): void { ... }
```

Comment when:

- The reason for a decision is non-obvious

- A workaround exists for an external library bug (include the bug link)
- The code is intentionally doing something that looks wrong

Don't comment when:

- The code explains itself
- The test explains the expected behavior

6.2 JSDoc / Docstring Format

```
/**
 * Creates a new user account and sends a verification email.
 *
 * @param dto - User creation data (email must be unique)
 * @returns The created user object, or a typed error
 * @throws {RateLimitError} If the creation rate limit is exceeded
 * @example
 * const result = await createUser({ email: 'user@example.com', name: 'Alice' });
 * if (!result.ok) handleError(result.error);
 */
async function createUser(dto: CreateUserDto): Promise<Result<User>> { ... }
```

6.3 README Requirements

Every repository/service MUST have a README with:

1. **What** — One sentence describing the service
2. **Quick start** — How to run it locally (3-5 commands)
3. **Key commands** — `dev`, `test`, `build`, `lint`
4. **Links** — Architecture doc, API docs, team channel

6.4 ADR Requirements

Write an Architecture Decision Record (ADR) when:

- Choosing a new major dependency
- Changing the tech stack
- Making a significant architectural change
- Explicitly choosing NOT to do something common

ADR location: `{{ADR_DIR}}/YYYY-MM-DD-decision-title.md` **ADR template:** See `{{ADR_TEMPLATE}}`

---

7. Security Coding Practices

Practice	Rule
Input validation	Validate ALL external input at system boundaries. Use schema validation (Zod / Joi / Pydantic)
SQL queries	NEVER use string interpolation in SQL. Always use parameterized queries / ORM
Authentication	NEVER implement auth from scratch. Use battle-tested libraries
Secrets	NEVER hardcode secrets. Use environment variables sourced from secret manager
Logging	NEVER log PII, passwords, tokens, or payment data
Dependencies	Review new dependencies for known CVEs before adding
Error messages	NEVER expose internal details (stack traces, DB errors) to users
HTML output	Always escape user-generated content. Use framework's built-in escaping
File uploads	Validate type, size, and content. Never execute uploaded files

Security review trigger: Any PR touching authentication, authorization, payments, or user data must be reviewed by {{SECURITY_REVIEWER}}.

---

8. Performance Coding Practices

Practice	Rule
Database queries	Avoid N+1 — use `include`/`JOIN` or batch loading. Profile with query analyzer
Pagination	Never load unbounded lists. Always paginate or stream
Caching	Cache expensive computations and external API calls. Define TTL explicitly
Async	Use async I/O for all I/O operations. Never block the event loop
Indexes	Add database indexes for any column used in `WHERE` or `ORDER BY` at scale
Large payloads	Compress API responses. Stream large files rather than buffering in memory

Related Documents

- Developer Onboarding Guide
- Local Development Setup
- Test Strategy
- Definition of Done

Approval

Role	Name	Date	Signature
Author			
Reviewer			
Approver			

Revision #3
Created 2026-02-24 14:54:32 UTC by John
Updated 2026-05-25 07:34:56 UTC by John