Git Commit Standards

This document defines the commit message conventions used in the BackTrade project. Following these standards ensures a clear, consistent, and maintainable git history that facilitates code reviews, automated changelog generation, and project maintenance.

Table of Contents

Commit Message Format

All commit messages MUST follow the **Conventional Commits** specification:

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

[optional body]

[optional footer(s)]
```

Format Rules

1. **Type** and **scope** are **required** for all commits
2. **Subject** must be in **lowercase** and use **imperative mood** (e.g., "fix bug" not "fixed bug" or "fixes bug")
3. **Subject** must not end with a period
4. **Subject** should be concise but descriptive (50-72 characters recommended)
5. **Type** and **scope** are separated by a colon and space: `type(scope):`
6. **Scope** is enclosed in parentheses and is optional but highly recommended

---

Commit Types

The following types are allowed in commit messages:

Type	Description	Example
`feat`	A new feature that adds functionality to the application	`feat(api): add user authentication`
`fix`	A bug fix that resolves an issue or incorrect behavior	`fix(web): resolve chart rendering issue`
`docs`	Documentation-only changes (README, code comments, API docs)	`docs(api): update authentication guide`
`style`	Code style changes (formatting, whitespace, semicolons) that do not affect functionality	`style(web): format component with prettier`
`refactor`	Code refactoring that neither fixes a bug nor adds a feature	`refactor(api): simplify user service`
`perf`	Performance improvements	`perf(api): optimize database queries`
`test`	Adding or updating tests	`test(api): add user service unit tests`
`chore`	Maintenance tasks, dependency updates, build configuration changes	`chore(deps): update express to v4.18`
`ci`	Changes to CI/CD configuration files	`ci: update GitHub Actions workflow`
`build`	Changes to build system or external dependencies	`build: update webpack configuration`
`revert`	Reverts a previous commit	`revert: "feat(api): add new endpoint"`

Type Selection Guidelines

- **Use** `feat` when adding new functionality that users or developers will interact with
- **Use** `fix` when correcting incorrect behavior or resolving bugs
- **Use** `chore` for dependency updates, tooling changes, or maintenance tasks
- **Use** `refactor` when restructuring code without changing behavior
- **Use** `docs` only for documentation changes (not code comments)

Scopes

Scopes identify the area of the codebase affected by the change. They should be **lowercase** and match the project structure.

Standard Scopes

Based on the BackTrade project structure, the following scopes are recommended:

Application Scopes

Scope	Description	Example
`api`	Backend API changes (Express, controllers, services, routes)	`fix(api): validate request body`
`web`	Frontend application changes (React components, pages, hooks)	`feat(web): add user dashboard`

Package Scopes

Scope	Description	Example
`types`	Shared TypeScript types and Zod schemas	`feat(types): add instrument type schema`
`utils`	Shared utility functions	`refactor(utils): optimize date formatting`
`datas`	Data access layer (Prisma, repositories)	`fix(datas): correct user repository query`
`config`	Configuration packages	`chore(config): update environment schema`

Infrastructure Scopes

Scope	Description	Example
`docker`	Docker configuration, Dockerfiles, docker-compose	`fix(docker): update minio image version`
`k8s`	Kubernetes manifests and configuration	`feat(k8s): add HPA for backend service`
`arch`	Architecture changes affecting multiple systems	`feat(arch): introduce SSL for minio`
`ci`	CI/CD pipeline configuration	`ci: add automated testing workflow`
`build`	Build system configuration	`build: update turbo configuration`

Other Scopes

Scope	Description	Example
`deps`	Dependency updates (when not app-specific)	`chore(deps): update all packages`
`docs`	Documentation files	`docs: add git commit standards`
`test`	Test configuration or shared test utilities	`test: update jest configuration`

Scope Selection Guidelines

1. **Be specific**: Use the most specific scope that accurately describes the change
2. **Match structure**: Scopes should align with the project's directory structure
3. **Use** `arch` for architectural changes that span multiple systems or affect infrastructure
4. **Avoid generic scopes**: Prefer specific scopes like `api` or `web` over generic ones like `app`
5. **Consistency**: Use consistent naming (e.g., always use `arch`)

Message Body

The message body is **optional** but recommended for complex changes. It should:

- Provide additional context about the change
- Explain **why** the change was made, not just **what** changed
- Reference related issues or pull requests
- Be separated from the subject by a blank line
- Wrap at 72 characters per line

Body Format

```
feat(api): add user authentication endpoint

Implement JWT-based authentication with refresh token support.
This enables secure user sessions and addresses security
requirements outlined in issue #123.

- Add POST /api/v1/auth/login endpoint
- Add POST /api/v1/auth/refresh endpoint
- Implement token validation middleware
```

Breaking Changes

Breaking changes MUST be indicated in the commit message footer using the `BREAKING CHANGE:` keyword or by appending `!` after the type/scope.

Format 1: Footer

```
feat(api): change authentication response format

BREAKING CHANGE: The authentication endpoint now returns
tokens in a nested object instead of flat structure.
Migration guide available in docs/auth-migration.md.
```

Format 2: Type Suffix

```
feat(api)!: change authentication response format

The authentication endpoint now returns tokens in a nested
object instead of flat structure.
```

Examples

Here are commits following best practices:

```
fix(api): prevent instrument creation with existing id

Add validation to reject instrument creation requests that
include an id field, as instruments should be auto-generated.
```

```
feat(arch): add script to generate third party licenses for linux

Create shell script to automate license file generation
for compliance requirements on Linux systems.
```

```
fix(docker): pin minio image to specific version

Update minio dockerfile to use version 1.2.3 instead of
latest tag for better reproducibility and stability.
```

```
fix(api): correct minio downloadFile return type to Buffer

The downloadFile function was incorrectly typed. Update
implementation and types to ensure it returns Buffer as
expected by downstream consumers.
```

```
feat(arch): introduce SSL support for minio

Configure minio with SSL/TLS certificates to enable secure
communication. This includes certificate generation and
nginx proxy configuration updates.
```

```
fix(api): validate search query schema and return 400 on parse error

Add validation middleware to ensure search query parameters
conform to expected schema. Return BadRequestError when
validation fails instead of allowing invalid queries.
```

Complex Example with Body

```
feat(api): add dataset upload endpoint with validation

Implement POST /api/v1/datasets/upload endpoint that allows
users to upload trading datasets. The endpoint includes:

- File size validation (max 100MB)
- Content type validation (CSV only)
- Automatic dataset parsing and storage
- Integration with MinIO for file storage

Closes #296
```
