VERSIONS

- v1.0.0
- v1.0.0-beta.4
- v1.0.0-beta.3
- v1.0.0-beta.2
- v1.0.0-beta.1
- v1.0.0-beta

LANGUAGES

PAGES(15)

- English
- Italian
- Spanish
- Français
- 日本語
- 한국어
- Português Brasileiro
- ไทย
- Indonesia
- Deutsch
- Polish
- Русский
- Հայերեն
- 繁體中文
- 简体中文

Conventional Commits

A specification for adding human and machine readable meaning to co
messages

Quick Summary Full Specification Contribute

Summary

As an open-source maintainer, squash feature branches onto `master` and write a standardized commit message while doing so.

The commit message should be structured as follows:

```
<type>[optional scope]: <description>

[optional body]

[optional footer]
```

The commit contains the following structural elements, to communicate intent to the consumers of your library:

1. **fix:** a commit of the *type* `fix` patches a bug in your codebase (this correlates with `PATCH` in semantic versioning).
2. **feat:** a commit of the *type* `feat` introduces a new feature to the codebase (this correlates with `MINOR` in semantic versioning).
3. **BREAKING CHANGE:** a commit that has the text `BREAKING CHANGE:` at the beginning of its optional body or footer section introduces a breaking API change (correlating with `MAJOR` in semantic versioning). A breaking change can be part of commits of any *type*. e.g., a `fix:`, `feat:` & `chore:` types would all be valid, in addition to any other *type*.
4. Others: commit *types* other than `fix:` and `feat:` are allowed, for example **@commitlint/config-conventional** (based on the **the Angular convention**) recommends `chore:`, `docs:`, `style:`, `refactor:`, `perf:`, `test:`, and others. We also recommend `improvement` for commits that improve a current implementation without adding a new feature or fixing a bug. Notice these types are not mandated by the conventional commits specification, and have no implicit effect in semantic versioning (unless they include a BREAKING CHANGE, which is NOT recommended). A scope may be provided to a commit's type, to provide additional contextual information and is contained

Examples

Commit message with description and breaking change in body

```
feat: allow provided config object to extend other configs

BREAKING CHANGE: `extends` key in config file is now used for extending other config fil
```

Commit message with no body

```
docs: correct spelling of CHANGELOG
```

Commit message with scope

```
feat(lang): added polish language
```

Commit message for a fix using an (optional) issue number.

```
fix: minor typos in code

see the issue for details on the typos fixed

fixes issue #12
```

Introduction

In software development, it's been my experience that bugs are most often introduced at the boundaries between applications. Unit testing works great for testing the interactions that an open-source maintainer knows about, but do a poor job of capturing all the interesting, often unexpected, ways that a community puts a library to use.

Anyone who has upgraded to a new patch version of a dependency, only to watch their application start throwing a steady stream of 500 errors, knows how important a readable commit history (and **ideally a well maintained CHANGELOG**) is to the ensuing forensic process.

software developers to describe in commit messages, features, fixes, and breaking changes that they make.

By introducing this convention, we create a common language that makes it easier to debug issues across project boundaries.

Specification

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in **RFC 2119**.

1. Commits MUST be prefixed with a type, which consists of a noun, `feat`, `fix`, etc., followed by a colon and a space.
2. The type `feat` MUST be used when a commit adds a new feature to your application or library.
3. The type `fix` MUST be used when a commit represents a bug fix for your application.
4. An optional scope MAY be provided after a type. A scope is a phrase describing a section of the codebase enclosed in parenthesis, e.g., `fix(parser):`
5. A description MUST immediately follow the type/scope prefix. The description is a short description of the code changes, e.g., *fix: array parsing issue when multiple spaces were contained in string*.
6. A longer commit body MAY be provided after the short description, providing additional contextual information about the code changes. The body MUST begin one blank line after the description.
7. A footer MAY be provided one blank line after the body (or after the description if body is missing). The footer SHOULD contain additional issue references about the code changes (such as the issues it fixes, e.g., `Fixes #13`).
8. Breaking changes MUST be indicated at the very beginning of the footer or body section of a commit. A breaking change MUST consist of the uppercase text `BREAKING CHANGE`, followed by a colon and a space.
9. A description MUST be provided after the `BREAKING CHANGE:` , describing what has changed about the API, e.g., *BREAKING CHANGE: environment variables now take precedence over config files.*
10. The footer MUST only contain `BREAKING CHANGE`, external links, issue references, and other meta-information.

Why Use Conventional Commits

- Automatically generating CHANGELOGs.
- Automatically determining a semantic version bump (based on the types of commits landed).
- Communicating the nature of changes to teammates, the public, and other stakeholders.
- Triggering build and publish processes.
- Making it easier for people to contribute to your projects, by allowing them to explore a more structured commit history.

FAQ

How should I deal with commit messages in the initial development phase?

We recommend that you proceed as if you've an already released product. Typically *somebody*, even if its your fellow software developers, is using your software. They'll want to know what's fixed, what breaks etc.

What do I do if the commit conforms to more than one of the commit types?

Go back and make multiple commits whenever possible. Part of the benefit of Conventional Commits is its ability to drive us to make more organized commits and PRs.

Doesn't this discourage rapid development and fast iteration?

It discourages moving fast in a disorganized way. It helps you be able to move fast long term across multiple projects with varied contributors.

Might Conventional Commits lead developers to limit the type of commits they make because they'll be thinking in the types provided?

Conventional Commits encourages us to make more of certain types of commits such as fixes. Other than that, the flexibility of Conventional Commits allows your team to come up with their own types and change those types over time.

translated to `MINOR` releases. Commits with `BREAKING CHANGE` in the commits, regardless of type, should be translated to `MAJOR` releases.

How should I version my extensions to the Conventional Commits Specification, e.g. `@jameswomack/conventional-commit-spec` ?

We recommend using SemVer to release your own extensions to this specification (and encourage you to make these extensions!)

What do I do if I accidentally use the wrong commit type?

When you used a type that's of the spec but not the correct type, e.g. `fix` instead of `feat`

Prior to merging or releasing the mistake, we recommend using `git rebase -i` to edit the commit history. After release, the cleanup will be different according to what tools and processes you use.

When you used a type *not* of the spec, e.g. `feet` instead of `feat`

In a worst case scenario, it's not the end of the world if a commit lands that does not meet the conventional commit specification. It simply means that commit will be missed by tools that are based on the spec.

Do all my contributors need to use the conventional commit specification?

No! If you use a squash based workflow on Git lead maintainers can clean up the commit messages as they're merged—adding no workload to casual committers. A common workflow for this is to have your git system automatically squash commits from a pull request and present a form for the lead maintainer to enter the proper git commit message for the merge.

About

The Conventional Commit specification is inspired by, and based heavily on, the **Angular Commit Guidelines**.

The first draft of this specification has been written in collaboration with some of the folks

git histories.

- **unleash**: a tool for automating the software release and publishing lifecycle.
- **lerna**: a tool for managing monorepos, which grew out of the Babel project.

Projects Using Conventional Commits

- **yargs**: everyone's favorite pirate themed command line argument parser.
- **parse-commit-message**: Spec compliant parsing utility to get object like `{ header: { type, scope, subject }, body, footer }` from given commit message string.
- **istanbuljs**: a collection of open-source tools and libraries for adding test coverage to your JavaScript tests.
- **standard-version**: Automatic versioning and CHANGELOG management, using GitHub's new squash button and the recommended Conventional Commits workflow.
- **uPortal-home** and **uPortal-application-framework**: Optional supplemental user interface enhancing **Apereo uPortal**.
- **Nintex Forms**: Easily create dynamic online forms to capture and submit accurate and current data.

want your project on this list? **send a pull request**.