# Conventional Commits 1.0.0

## Summary

The Conventional Commits specification is a lightweight convention on top of commit messages. It provides an easy set of rules for creating an explicit commit history; which makes it easier to write automated tools on top of. This convention dovetails with SemVer, by describing the features, fixes, and breaking changes made in commit messages.

The commit message should be structured as follows:

---

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

[optional body]

[optional footer(s)]
```

---

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 a footer `BREAKING CHANGE:`, or appends a `!` after the type/scope, introduces a breaking API change (correlating with `MAJOR` in Semantic Versioning). A BREAKING CHANGE can be part of commits of any *type*.
4. *types* other than `fix:` and `feat:` are allowed, for example **@commitlint/config-conventional** (based on the **the Angular convention**) recommends `build:`, `chore:`, `ci:`, `docs:`, `style:`, `refactor:`, `perf:`, `test:`, and others.
5. *footers* other than `BREAKING CHANGE: <description>` may be provided and follow a convention similar to **git trailer format**.

Additional types are not mandated by the Conventional Commits specification, and have no implicit effect in Semantic Versioning (unless they include a BREAKING CHANGE). A scope may be provided to a commit's type, to provide additional contextual information and is contained within parenthesis, e.g., `feat(parser): add ability to parse arrays`.

## Examples

### Commit message with description and breaking change footer

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

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

### Commit message with `!` to draw attention to breaking change

```
feat!: send an email to the customer when a product is shipped
```

### Commit message with scope and `!` to draw attention to breaking change

```
feat(api)!: send an email to the customer when a product is shipped
```

### Commit message with both `!` and BREAKING CHANGE footer

```
chore!: drop support for Node 6

BREAKING CHANGE: use JavaScript features not available in Node 6.
```

### Commit message with no body

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

### Commit message with scope

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

### Commit message with multi-paragraph body and multiple footers

```
fix: prevent racing of requests

Introduce a request id and a reference to latest request. Dismiss
incoming responses other than from latest request.

Remove timeouts which were used to mitigate the racing issue but are
obsolete now.

Reviewed-by: Z
Refs: #123
```

## 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 the OPTIONAL scope, OPTIONAL `!`, and REQUIRED terminal colon and 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. A scope MAY be provided after a type. A scope MUST consist of a noun describing a section of the codebase surrounded by parenthesis, e.g., `fix(parser):`
5. A description MUST immediately follow the colon and space after the type/scope prefix. The description is a short summary 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 commit body is free-form and MAY consist of any number of newline separated paragraphs.
8. One or more footers MAY be provided one blank line after the body. Each footer MUST consist of a word token, followed by either a `:<space>` or `<space>#` separator, followed by a string value (this is inspired by the **git trailer convention**).
9. A footer's token MUST use `-` in place of whitespace characters, e.g., `Acked-by` (this helps differentiate the footer section from a multi-paragraph body). An exception is made for `BREAKING CHANGE`, which MAY also be used as a token.
10. A footer's value MAY contain spaces and newlines, and parsing MUST terminate when the next valid footer token/separator pair is observed.
11. Breaking changes MUST be indicated in the type/scope prefix of a commit, or as an entry in the footer.
12. If included as a footer, a breaking change MUST consist of the uppercase text BREAKING CHANGE, followed by a colon, space, and description, e.g., *BREAKING CHANGE: environment variables now take precedence over config files*.
13. If included in the type/scope prefix, breaking changes MUST be indicated by a `!` immediately before the `:`. If `!` is used, `BREAKING CHANGE:` MAY be omitted from the footer section, and the commit description SHALL be used to describe the breaking change.
14. Types other than `feat` and `fix` MAY be used in your commit messages, e.g., *docs: updated ref docs*.
15. The units of information that make up Conventional Commits MUST NOT be treated as case sensitive by implementors, with the exception of BREAKING CHANGE which MUST be uppercase.
16. BREAKING-CHANGE MUST be synonymous with BREAKING CHANGE, when used as a token in a footer.

## 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 already released the product. Typically *somebody*, even if it's your fellow software developers, is using your software. They'll want to know what's fixed, what breaks etc.

### Are the types in the commit title uppercase or lowercase?

Any casing may be used, but it's best to be consistent.

### 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.

### How does this relate to SemVer?

`fix` type commits should be translated to `PATCH` releases. `feat` type commits should be 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 Commits 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 Commits 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.

### How does Conventional Commits handle revert commits?

Reverting code can be complicated: are you reverting multiple commits? if you revert a feature, should the next release instead be a patch?

Conventional Commits does not make an explicit effort to define revert behavior. Instead we leave it to tooling authors to use the flexibility of *types* and *footers* to develop their logic for handling reverts.

One recommendation is to use the `revert` type, and a footer that references the commit SHAs that are being reverted:

```
revert: let us never again speak of the noodle incident

Refs: 676104e, a215868
```