Understanding Semantic Commit Messages Using Git and Angular

December 21, 2019  10 min read

*Let's explain the Semantic Commits term and demonstrate practical examples of commit messages, inspired by the Conventional Commit specification and Angular conventions.*

Many projects decide to standardize their commit messages with conventions, one way or the other. This practice isn’t new, but increasingly applied in the last few years. Most likely you’ve already encountered such commit messages in certain projects.

One of the first specifications that have come up belongs to the AngularJS project. The team created a detailed [document](https://docs.google.com/document/d/1QrDFcIiPjSLDn3EL15IJygNPiHORgU1_OOAqWjiDU5Y/edit) which specifies the goals and way they’re supposed to commit. These commit conventions are pretty popular, and some of you maybe reached them through [Karma](http://karma-runner.github.io/4.0/dev/git-commit-msg.html) guidelines. And yet, there are also different convention variations as of [jQuery](https://contribute.jquery.org/commits-and-pull-requests/#commit-guidelines), [JSHint](https://github.com/jshint/jshint/blob/master/CONTRIBUTING.md" \l "commit-message-guidelines" \t "_blank), [Ember](https://github.com/emberjs/ember.js/blob/master/CONTRIBUTING.md#commit-tagging), [Angular](https://github.com/angular/angular/blob/master/CONTRIBUTING.md#-commit-message-guidelines) (an enhanced version that’s inspired by the AngularJS commit specification) and even [more](https://github.com/conventional-changelog/conventional-changelog/tree/master/packages):

一張含有 文字 的圖片

自動產生的描述

Commit convention variations

It’s plain to see the variety of commit conventions above, which definitely constitutes a decent reason to standardize an official specification. [Conventional Commits](https://www.conventionalcommits.org/) is such a specification, which in practice, simplifies the Angular conventions and lightly specifies the essentials of commit message conventions.

During this article, we’ll introduce the idea behind “Semantic Commits” and demonstrate concrete examples, using Git and the Angular commit conventions. For the record, we use them only to clarify the concept - which obviously means the version control tool and the specification is up to you.

Here we go! 👨🏻‍🏫

Motivation

Let’s start by defining the term in general:

*Semantic Commits are commit messages with human and machine readable meaning, which follow particular conventions*

This means, it’s is merely guidelines for **commit messages**, so that:

* The commit messages are **semantic** - because these are categorized into meaningful types, indicating the essence of the commit
* The commit messages are **conventional** - because these are formatted by a consistent structure and well-known types, both for developers and tools

Further to that, semantic commits might come in handy when we typically need to:

1. Allow the maintainers and contributors to easily browse the project history and understand the essence of changes, while ignoring unimportant changes by commit message type
2. Enforce restricted commit structure, thereby encouraging smaller commits with a specific purpose
3. Commit the message subject directly, without messing with the wording
4. Bump the package version automatically, based on commit message types
5. Generate CHANGELOGs and release notes automatically

To wrap up, semantic commits are dedicated to achieving better **readability**, **velocity** and **automation**.

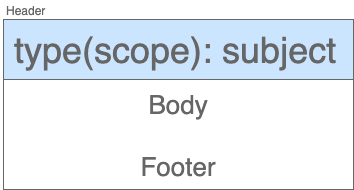
Having said that, it definitely makes sense that some of us might not accept these message conventions as readable or informative. In case we also don’t need the additional benefits that arrive with them - it’s apparently senseless to enforce such a specification within the project.

Alright, it’s about time to understand how we practically follow the conventions.

***Disclaimer:****From this moment on, we’re going to refer the Angular commit message conventions and their benefits.*

Commit Message Format

The Angular conventions demand to shape the commit message according to the following structure:



The commit message consists of a header, body and footer

The diagram above illustrates to us that the commit message consists of three parts - **header**, **body** and **footer**.

Let’s elaborate on each part.

The Header

The header is a **mandatory** line that simply describes the purpose of the change (up to 100 characters).

Better yet, it consists of three parts in itself:

1. Type - a short prefix that represents the **kind** of the change
2. Scope - optional information (attached to the prefix) that represents the **context** of the change
3. Subject - represents a concise **description** of the actual change

Practically, in terms of Git, it’s merely the first line of the commit message:

|  |
| --- |
| git commit -m "fix(core): remove deprecated and defunct wtf\* apis" |

We insert a single-line message which is separated by :. The left partition is what we hypothetically named “prefix” - when fix and core (the affected package) are the type and scope respectively. On the other hand, the right partition obviously constitutes the subject.

Simply put, the above message meaning is - *“This change fixes a bug that belongs to the Core package, by removing deprecated and defunct wtf\* apis”*.

The Body

The body is **optional** lines that introduce the motivation behind the change or just describing slightly more detailed information.

Let’s take the recent example and add a body:

|  |
| --- |
| git commit -m "fix(core): remove deprecated and defunct wtf\* apis" -m "These apis have been deprecated in v8, so they should stick around till v10, but since they are defunct we are removing them early so that they don't take up payload size." |

Now we attach to the message a couple of sentences that explain the purpose in detail.

Notice the following:

* We use multiple -m in order to concatenate **paragraphs** instead of simple lines
* The header and body are **supposed** to be separated by a blank line (and that’s distinctly true due to the paragraphs)

***Note:****Although we could use other ways to break the message into lines - we’ll keep using multiple -m in the next examples for the simplicity (and also to show a shell-agnostic solution).*

The Footer

The footer is **optional** lines that mention consequences which stems from the change - such as announcing a breaking change, linking closed issues, mentioning contributors and so on.

Here’s the recent commit message with a footer:

|  |
| --- |
| git commit -m "fix(core): remove deprecated and defunct wtf\* apis" -m "These apis have been deprecated in v8, so they should stick around till v10, but since they are defunct we are removing them early so that they don't take up payload size." -m "PR Close #33949" |

In this case, we plainly add a reference to the relevant pull request and nothing else.

To finish, let’s view the complete commit log:

一張含有 文字 的圖片

自動產生的描述

Showing the commit log in one piece

As you might infer, this [commit](https://github.com/angular/angular/commit/cf420194ed91076afb66d9179245b9dbaabc4fd4) was actually made in the Angular repository.

Common Types

On top of defining the commit message format, the Angular commit message conventions specify a list of useful types that cover various sorts of changes.

Before we begin, we should distinguish between two categories **of** types:

* Development - sort of maintenance types which classify changes, intended the **developers**, that don’t actually affect the production code but rather the development environment and workflow internally
* Production - sort of enhancement types which classify changes, intended the **end users**, that solely affect the production code

Now, let’s introduce and explain these handy types.

***Note:****The examples below are taken directly from the commit log of the Angular repository.*

👷 build

The build type (formerly known as chore) is used to identify **development** changes related to the build system (involving scripts, configurations or tools) and package dependencies.

Examples:

一張含有 文字 的圖片

自動產生的描述

Examples of commit messages with `build` type

💚 ci

The ci type is used to identify **development** changes related to the continuous integration and deployment system - involving scripts, configurations or tools.

Examples:

一張含有 文字 的圖片

自動產生的描述

Examples of commit messages with `ci` type

📝 docs

The docs type is used to identify documentation changes related to the project - whether intended externally for the end users (in case of a library) or internally for the developers.

Examples:

一張含有 文字 的圖片

自動產生的描述

Examples of commit messages with `docs` type

✨ feat

The feat type is used to identify **production** changes related to new backward-compatible abilities or functionality.

Examples:

一張含有 文字 的圖片

自動產生的描述

Examples of commit messages with `feat` type

🐛 fix

The fix type is used to identify **production** changes related to backward-compatible bug fixes.

Examples:

一張含有 文字 的圖片

自動產生的描述

Examples of commit messages with `fix` type

⚡️ perf

The perf type is used to identify **production** changes related to backward-compatible performance improvements.

Examples:

一張含有 文字 的圖片

自動產生的描述

Examples of commit messages with `perf` type

♻️ refactor

The refactor type is used to identify **development** changes related to modifying the codebase, which neither adds a feature nor fixes a bug - such as removing redundant code, simplifying the code, renaming variables, etc.

Examples:

一張含有 文字 的圖片

自動產生的描述

Examples of commit messages with `refactor` type

🎨 style

The style type is used to identify **development** changes related to styling the codebase, regardless of the meaning - such as indentations, semi-colons, quotes, trailing commas and so on.

Examples:

一張含有 文字 的圖片

自動產生的描述

Examples of commit messages with `style` type

✅ test

The test type is used to identify **development** changes related to tests - such as refactoring existing tests or adding new tests.

Examples:

一張含有 文字 的圖片

自動產生的描述

Examples of commit messages with `test` type

Benefits

Now that we’re familiar with the conventions - let’s see two ways to benefit from them.

Browsing History

Git provides us the power to browse the repository commit history - so we’re able to figure out what actually happened, who contributed and so on.

Let’s see how the conventions might ease up the browsing:

|  |
| --- |
| git log --oneline --grep "^feat\|^fix\|^perf" |

We use the commit message type to filter out and so showing only the production changes (all of the messages that start with feat, fix or perf).

Another example:

|  |
| --- |
| git log --oneline --grep "^feat" | wc -l |

We just print the total amount of feat changes.

The point is - the commit message format is very structured, what effectively allows us relying on that when scanning or filtering the commit history.

Namely, a better velocity! 💪🏻

Automated Releases

The commit message format is either useful for automating steps of the release process.

In fact, this is possible due to tools like [Standard Version](https://github.com/conventional-changelog/standard-version) and [Semantic Release](https://github.com/semantic-release/semantic-release) that strictly following the [Semantic Versioning](https://semver.org/) specification beside certain commit message conventions (Conventional Commits and Angular conventions respectively). The main difference between them is the [approach](https://github.com/conventional-changelog/standard-version#how-is-standard-version-different-from-semantic-release), but let’s focus on Semantic Release.

So, based on the commit message (and especially the type) - Semantic Release is able to:

* Bump to the next semantic package version (when fix causes to **patch**, feat & perf to **minor**, and obviously - breaking change to **major**)
* Generate a CHANGELOG file and release notes containing the relevant **production** changes
* Create a Git tag for the new release version
* Publish the release artifact into an npm registry

That’s pretty cool, right?

Ionic’s [angular-toolkit](https://github.com/ionic-team/angular-toolkit) project, for instance, integrates Semantic Release to automate the release process (hereby follows the Angular commit conventions):

一張含有 文字 的圖片

自動產生的描述

An example of a generated release version

As we notice, a release version was generated with the correct tag and notes - but the thing is, that was done automatically. 🤖

Miscellaneous

Let’s see a couple of stuff in order to make the most of semantic commits.

Using Emojis

Attaching Emojis to the commit message might improve the readability even more, so that we can identify them pretty quickly and easily while browsing the commit history. 💯

Check out the following links:

* [gitmoji](https://gitmoji.carloscuesta.me/)
* [Commit Message Emoji 👋](https://github.com/dannyfritz/commit-message-emoji)

CLI Tool

[Commitizen](https://github.com/commitizen/cz-cli) is a tool that enables to enforce a commit message format using the command line:

一張含有 文字 的圖片

自動產生的描述

Semantic commit message using the command line

Linter

[commitlint](https://github.com/conventional-changelog/commitlint) is a tool that guarantees which the commit message format is aligned with the conventions:

一張含有 文字 的圖片

自動產生的描述

Linting the commit messages

VS Code Extension

If you’d like to use a customisable [VS Code extension](https://github.com/nitayneeman/vscode-git-semantic-commit), then the following might interest you:

Semantic commit message using the Command Palette

Honestly, it was developed by myself. 😊

Summary

We introduced today the term of “Semantic Commits” and explained the structure of such a message, through concrete examples that follow the Angular commit message conventions.

Recapping the top points:

* Semantic commits are commit messages with a meaning, both for developers and tools, which follow particular conventions
* Semantic commits (along with its based-on tools) help to improve the readability, velocity and automation
* Conventional Commits is a specification that details semantic commits which follow lightweight conventions
* Angular’s guidelines detail semantic commits which follow the project conventions, including:
  + A message format containing header, body and footer
  + Types of commit changes, related to development and production
* We can benefit from the message conventions to browse the commit history easily
* We can benefit from the message conventions to automate the release process

And finally, whether you decide to adopt such conventions or not - you might keep encountering them occasionally, so just keep in mind the points above. 😉