commit-message-format

Contents

- Commit Message Guidelines
  - Why?
  - Commit Message Format
    - `<type>`
    - `<scope>`
    - `<subject>`
    - `<body>`
    - `<footer>`
  - Generate a CHANGELOG
  - Ignore non-important commits
    - Examples
  - Pull Requests

Commit Message Guidelines

This document gathers wisdom from different sources to provide a sensible guide for writing commit messages.

Why?

- provide a structured and clear project information while browsing git history
- allow an easy generation of the `CHANGELOG` by a script
- make it easy to ignore non-important commits

Commit Message Format

A commit message consists of a **header**, a **body** and a **footer**. The header has a special format that includes a **type**, an optional **scope** and a **subject**:

`<type>(<scope>): <subject> <BLANK LINE> <body> <BLANK LINE> <footer>`

The **header** is mandatory and the **scope** element of the header is optional.

The footer should contain a closing reference to an issue if any.

Many projects sticks to the 50/72 rule by convention

- maximum 50 characters long first line (excluding type and scope)
- description maximum 72 lines long

In general the following guideline can be followed

- Make separate commits for logically separate changes.
- Describe your changes well.

`<type>`

Project defined, but some more common examples could be:

- **build**: Changes that affect the build system or external dependencies (example scopes: gulp, broccoli, npm)
- **ci**: Changes to our CI configuration files and scripts (example scopes: Circle, BrowserStack, SauceLabs)
- **docs**: Documentation only changes
- **feat**: A new feature (or **feature**)
- **license**: Licensing compliance and changes
- **fix**: A bug fix
- **perf**: A code change that improves performance
- **refactor**: A code change that neither fixes a bug nor adds a feature
- **style**: Changes that do not affect the meaning of the code (white-space, formatting, missing semi-colons, etc)
- **test**: Adding missing tests or correcting existing tests
- **revert**: If the commit reverts a previous commit, it should begin with `revert:`, followed by the header of the reverted commit. In the body it should say: `This reverts commit <hash>.`, where the hash is the SHA of the commit being reverted.

`<scope>`

If applicable, the scope should be the name of the package affected (as perceived by the person reading the changelog generated from commit messages). The following is a list of example scopes:

- **common**
- **compiler**
- **compiler-cli**
- **core**
- **http**

- **router**

`<subject>`

The subject contains a succinct description of the change:

- use the imperative, present tense: “change” not “changed” nor “changes”. Read your commit message as “This commit will ..”
- don’t capitalize the first letter
- no dot (.) at the end

`<body>`

Just as in the **subject**, use the imperative, present tense: “change” not “changed” nor “changes”. The body should include the motivation for the change and contrast this with previous behavior.

`<footer>`

The footer should contain any information about **Breaking Changes** and is also the place to reference issue numbers or identifiers that this commit **Closes**.

1. Breaking changes - Start with the word `BREAKING CHANGE:` with a space or two newlines. The rest of the commit message is then used for this.
2. Referencing issues - Closed bugs should be listed on a separate line in the footer prefixed with “Closes” keyword like this: `Closes #234` or in case of multiple issues: `Closes #123, #245, #992`

Generate a CHANGELOG

Use these three sections in a changelog: new features, bug fixes, breaking changes. This list could be generated by a script when doing a release, along with links to related commits.

1. List of all subjects (first lines in commit message) since last release:

   ```
   bash git log <last tag> HEAD --pretty=format:%s
   ```

2. List only the new New features in this release

   ```
   bash git log <last release> HEAD --grep feat
   ```

Ignore non-important commits

You might want to ignore certain commits, like formatting changes For example, when bisecting, you can ignore these by:

```
bash git bisect skip $(git rev-list --grep irrelevant <good place> HEAD)
```

Examples

```
docs: update changelog to beta
```

```
fix(pencil): stop graphite breaking when too much pressure applied
```

```
feat(pencil): add 'graphiteWidth' option
```

```
perf(pencil): remove graphiteWidth option

BREAKING CHANGE: The graphiteWidth option has been removed.
```

```
The default graphite width of 10mm is always used for performance reasons.
(Note that the BREAKING CHANGE:  token must be in the footer of the commit)
```

```
feat(directive): add directives disabled/checked/multiple/readonly

New directives for proper binding these attributes in older browsers. Added
   coresponding description, live examples and
e2e tests.

Closes #351
```

```
feat(compile): simplify isolate scope bindings

Change the isolate scope binding options to:
- @attr - attribute binding (including interpolation)
- =model - by-directional model binding
- &expr - expression execution binding

This change simplifies the terminology as well as number of choices available
    to the developer. It also supports local
name aliasing from the parent.

BREAKING CHANGE: isolate scope bindings definition has changed and the inject
    option for the directive controller
injection was removed.

To migrate the code follow the example below:

Before:

scope: { myAttr: 'attribute', myBind: 'bind', myExpression: 'expression',
   myEval: 'evaluate', myAccessor: 'accessor' }

After:

scope: { myAttr: '@', myBind: '@', myExpression: '&', // myEval - usually not
    useful, but in cases where the expression
is assignable, you can use '=' myAccessor: '=' // in directive's template
   change myAccessor() to myAccessor }

The removed `inject` wasn't generaly useful for directives so there should be
    no code using it.
```

Pull Requests

- To resolve conflicts, rebase pull request branches onto their target branch instead of merging the target branch into the pull request branch. This again results in a cleaner history without “criss-cross” merges.
- When addressing review comments in a pull request, please fix the issue in the commit where it appears, not in a new commit on top of the pull request’s history. While this requires force-pushing of the new iteration of your pull request’s branch, it has several advantages:

- Reviewers that go through (larger) pull requests commit by commit are always up-to-date with latest fixes, instead of coming across a commit that addresses their remarks only at the end.
- It maintains a cleaner history without distracting commits like "Address review comments".
- As a result, tools like git-bisect can operate in a more meaningful way.
- Fixing up commits allows for making fixes to commit messages, which is not possible by only adding new commits.

  If you are unfamiliar with fixing up existing commits, please read about rewriting history and `git rebase --interactive` in particular.