Contributing to Angular

We would love for you to contribute to Angular and help make it even better than it is today! As a contributor, here are the guidelines we would like you to follow:

Code of Conduct

Help us keep Angular open and inclusive. Please read and follow our Code of Conduct.

Got a Question or Problem?

Do not open issues for general support questions as we want to keep GitHub issues for bug reports and feature requests. Instead, we recommend using Stack Overflow to ask support-related questions. When creating a new question on Stack Overflow, make sure to add the `angular` tag.

Stack Overflow is a much better place to ask questions since:

- there are thousands of people willing to help on Stack Overflow
- questions and answers stay available for public viewing so your question/answer might help someone else
- Stack Overflow's voting system assures that the best answers are prominently visible.

To save your and our time, we will systematically close all issues that are requests for general support and redirect people to Stack Overflow.

If you would like to chat about the question in real-time, you can reach out via our Discord server.

Found a Bug?

If you find a bug in the source code, you can help us by submitting an issue to our GitHub Repository. Even better, you can submit a Pull Request with a fix.

Missing a Feature?

You can *request* a new feature by submitting an issue to our GitHub Repository. If you would like to *implement* a new feature, please consider the size of the change in order to determine the right steps to proceed:

- For a **Major Feature**, first open an issue and outline your proposal so that it can be discussed. This process allows us to better coordinate our efforts, prevent duplication of work, and help you to craft the change so that it is successfully accepted into the project.

  Note: Adding a new topic to the documentation, or significantly re-writing a topic, counts as a major feature.

- **Small Features** can be crafted and directly submitted as a Pull Request.

Submission Guidelines

Submitting an Issue

Before you submit an issue, please search the issue tracker. An issue for your problem might already exist and the discussion might inform you of workarounds readily available.

We want to fix all the issues as soon as possible, but before fixing a bug, we need to reproduce and confirm it. In order to reproduce bugs, we require that you provide a minimal reproduction. Having a minimal reproducible scenario gives us a wealth of important information without going back and forth to you with additional questions.

A minimal reproduction allows us to quickly confirm a bug (or point out a coding problem) as well as confirm that we are fixing the right problem.

We require a minimal reproduction to save maintainers' time and ultimately be able to fix more bugs. Often, developers find coding problems themselves while preparing a minimal reproduction. We understand that sometimes it might be hard to extract essential bits of code from a larger codebase but we really need to isolate the problem before we can fix it.

Unfortunately, we are not able to investigate / fix bugs without a minimal reproduction, so if we don't hear back from you, we are going to close an issue that doesn't have enough info to be reproduced.

You can file new issues by selecting from our new issue templates and filling out the issue template.

Submitting a Pull Request (PR)

Before you submit your Pull Request (PR) consider the following guidelines:

1. Search GitHub for an open or closed PR that relates to your submission. You don't want to duplicate existing efforts.
2. Be sure that an issue describes the problem you're fixing, or documents the design for the feature you'd like to add. Discussing the design upfront helps to ensure that we're ready to accept your work.
3. Please sign our Contributor License Agreement (CLA) before sending PRs. We cannot accept code without a signed CLA. Make sure you author all contributed Git commits with email address associated with your CLA signature.
4. Fork the angular/angular repo.
5. In your forked repository, make your changes in a new git branch:

   ```
   git checkout -b my-fix-branch master
   ```

6. Create your patch, **including appropriate test cases**.
7. Follow our Coding Rules.
8. Run the full Angular test suite, as described in the developer documentation, and ensure that all tests pass.
9. Commit your changes using a descriptive commit message that follows our commit message conventions. Adherence to these conventions is necessary because release notes are automatically generated from these messages.

   ```
   git commit --all
   ```

Note: the optional commit `-a` command line option will automatically "add" and "rm" edited files.

10. Push your branch to GitHub:

    ```
    git push origin my-fix-branch
    ```

11. In GitHub, send a pull request to `angular:master`.

Reviewing a Pull Request

The Angular team reserves the right not to accept pull requests from community members who haven't been good citizens of the community. Such behavior includes not following the Angular code of conduct and applies within or outside of Angular managed channels.

Addressing review feedback

If we ask for changes via code reviews then:

1. Make the required updates to the code.
2. Re-run the Angular test suites to ensure tests are still passing.
3. Create a fixup commit and push to your GitHub repository (this will update your Pull Request):

    ```
    git commit --all --fixup HEAD
    git push
    ```

    For more info on working with fixup commits see here.

That's it! Thank you for your contribution!

Updating the commit message

A reviewer might often suggest changes to a commit message (for example, to add more context for a change or adhere to our commit message guidelines). In order to update the commit message of the last commit on your branch:

1. Check out your branch:

    ```
    git checkout my-fix-branch
    ```

2. Amend the last commit and modify the commit message:

    ```
    git commit --amend
    ```

3. Push to your GitHub repository:

    ```
    git push --force-with-lease
    ```

> *NOTE:*
> *If you need to update the commit message of an earlier commit, you can use* `git rebase` *in interactive mode.*

> *See the git docs for more details.*

After your pull request is merged

After your pull request is merged, you can safely delete your branch and pull the changes from the main (upstream) repository:

- Delete the remote branch on GitHub either through the GitHub web UI or your local shell as follows:

  ```
  git push origin --delete my-fix-branch
  ```

- Check out the master branch:

  ```
  git checkout master -f
  ```

- Delete the local branch:

  ```
  git branch -D my-fix-branch
  ```

- Update your master with the latest upstream version:

  ```
  git pull --ff upstream master
  ```

Coding Rules

To ensure consistency throughout the source code, keep these rules in mind as you are working:

- All features or bug fixes **must be tested** by one or more specs (unit-tests).
- All public API methods **must be documented**.
- We follow Google's JavaScript Style Guide, but wrap all code at **100 characters**.

  An automated formatter is available, see DEVELOPER.md.

Commit Message Format

This specification is inspired by and supersedes the AngularJS commit message format.

We have very precise rules over how our Git commit messages must be formatted. This format leads to **easier to read commit history**.

Each commit message consists of a **header**, a **body**, and a **footer**.

```
<header>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

The `header` is mandatory and must conform to the Commit Message Header format.

The `body` is mandatory for all commits except for those of type "docs". When the body is present it must be at least 20 characters long and must conform to the Commit Message Body format.

The `footer` is optional. The Commit Message Footer format describes what the footer is used for and the structure it must have.

Commit Message Header

```
<type>(<scope>): <short summary>
  │       │             │
  │       │             └⫸ Summary in present tense. Not capitalized. No period at
the end.
  │       │
  │       └⫸ Commit Scope: animations|bazel|benchpress|common|compiler|compiler-
cli|core|
  │                          elements|forms|http|language-service|localize|platform-
browser|
  │                          platform-browser-dynamic|platform-server|router|service-
worker|
  │                          upgrade|zone.js|packaging|changelog|docs-
infra|migrations|ngcc|ve|
  │                          devtools
  │
  └⫸ Commit Type: build|ci|docs|feat|fix|perf|refactor|test
```

The `<type>` and `<summary>` fields are mandatory, the `(<scope>)` field is optional.

Type

Must be one of the following:

- **build**: Changes that affect the build system or external dependencies (example scopes: gulp, broccoli, npm)
- **ci**: Changes to our CI configuration files and scripts (examples: CircleCi, SauceLabs)
- **docs**: Documentation only changes
- **feat**: A new feature
- **fix**: A bug fix
- **perf**: A code change that improves performance
- **refactor**: A code change that neither fixes a bug nor adds a feature
- **test**: Adding missing tests or correcting existing tests

Scope

The scope should be the name of the npm package affected (as perceived by the person reading the changelog generated from commit messages).

The following is the list of supported scopes:

- `animations`
- `bazel`
- `benchpress`
- `common`
- `compiler`
- `compiler-cli`
- `core`

- `elements`
- `forms`
- `http`
- `language-service`
- `localize`
- `platform-browser`
- `platform-browser-dynamic`
- `platform-server`
- `router`
- `service-worker`
- `upgrade`
- `zone.js`

There are currently a few exceptions to the "use package name" rule:

- `packaging` : used for changes that change the npm package layout in all of our packages, e.g. public path changes, package.json changes done to all packages, d.ts file/format changes, changes to bundles, etc.
- `changelog` : used for updating the release notes in CHANGELOG.md
- `dev-infra` : used for dev-infra related changes within the directories /scripts and /tools
- `docs-infra` : used for docs-app (angular.io) related changes within the /aio directory of the repo
- `migrations` : used for changes to the `ng update` migrations.
- `ngcc` : used for changes to the Angular Compatibility Compiler
- `ve` : used for changes specific to ViewEngine (legacy compiler/renderer).
- `devtools` : used for changes in the browser extension.
- none/empty string: useful for `test` and `refactor` changes that are done across all packages (e.g. `test: add missing unit tests`) and for docs changes that are not related to a specific package (e.g. `docs: fix typo in tutorial`).

Summary

Use the summary field to provide a succinct description of the change:

- use the imperative, present tense: "change" not "changed" nor "changes"
- don't capitalize the first letter
- no dot (.) at the end

Commit Message Body

Just as in the summary, use the imperative, present tense: "fix" not "fixed" nor "fixes".

Explain the motivation for the change in the commit message body. This commit message should explain *why* you are making the change. You can include a comparison of the previous behavior with the new behavior in order to illustrate the impact of the change.

Commit Message Footer

The footer can contain information about breaking changes and deprecations and is also the place to reference GitHub issues, Jira tickets, and other PRs that this commit closes or is related to. For example:

```
BREAKING CHANGE: <breaking change summary>
<BLANK LINE>
<breaking change description + migration instructions>
<BLANK LINE>
<BLANK LINE>
Fixes #<issue number>
```

or

```
DEPRECATED: <what is deprecated>
<BLANK LINE>
<deprecation description + recommended update path>
<BLANK LINE>
<BLANK LINE>
Closes #<pr number>
```

Breaking Change section should start with the phrase "BREAKING CHANGE: " followed by a summary of the breaking change, a blank line, and a detailed description of the breaking change that also includes migration instructions.

Similarly, a Deprecation section should start with "DEPRECATED: " followed by a short description of what is deprecated, a blank line, and a detailed description of the deprecation that also mentions the recommended update path.

Revert commits

If the commit reverts a previous commit, it should begin with `revert:` , followed by the header of the reverted commit.

The content of the commit message body should contain:

- information about the SHA of the commit being reverted in the following format: `This reverts commit <SHA>` ,
- a clear description of the reason for reverting the commit message.

Signing the CLA

Please sign our Contributor License Agreement (CLA) before sending pull requests. For any code changes to be accepted, the CLA must be signed. It's a quick process, we promise!

- For individuals, we have a simple click-through form.
- For corporations, we'll need you to print, sign and one of scan+email, fax or mail the form.

If you have more than one GitHub accounts, or multiple email addresses associated with a single GitHub account, you must sign the CLA using the primary email address of the GitHub account used to author Git commits and send pull requests.

The following documents can help you sort out issues with GitHub accounts and multiple email addresses:

- https://help.github.com/articles/setting-your-commit-email-address-in-git/
- https://stackoverflow.com/questions/37245303/what-does-usera-committed-with-userb-13-days-ago-on-github-mean
- https://help.github.com/articles/about-commit-email-addresses/
- https://help.github.com/articles/blocking-command-line-pushes-that-expose-your-personal-email-address/