# Contributing to RxJS

[Read and abide by the Code of Conduct](CODE\_OF\_CONDUCT.md)! Even if you don't read it,

it still applies to you. Ignorance is not an exemption.

Contents

- [Submitting a Pull Request (PR)](#submitting-a-pull-request-pr)

- [After your pull request is merged](#after-your-pull-request-is-merged)

- [Coding Style Guidelines](#coding-style-guidelines)

- [Documentation](#documentation)

- [Unit Tests](#unit-tests)

- [CI Tests](#ci-tests)

- [Performance Tests](#performance-tests)

- [Macro](#macro)

- [Micro](#micro)

- [Commit Message Guidelines](#commit-message-guidelines)

- [Commit Message Format](#commit-message-format)

- [Revert](#revert)

- [Type](#type)

- [Scope](#scope)

- [Subject](#subject)

- [Body](#body)

- [Footer](#footer)

---

- Related documents

- [Creating Operators](doc/operator-creation.md)

- [Writing Marble Tests](doc/writing-marble-tests.md)

---

(This document is a work in progress and is subject to change)

## Submitting a Pull Request (PR)

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

- Search [GitHub](https://github.com/ReactiveX/RxJS/pulls) for an open or closed PR

that relates to your submission. You don't want to duplicate effort.

- Make your changes in a new git branch:

```shell

git checkout -b my-fix-branch master

```

- Create your patch, following [code style guidelines](#coding-style-guidelines), and \*\*including appropriate test cases\*\*.

- Run the full test suite and ensure that all tests pass.

- Run the micro and macro performance tests against your feature branch and compare against master

to ensure performance wasn't changed for the worse.

- Commit your changes using a descriptive commit message that follows our

[commit message guidelines](#commit-message-guidelines). Adherence to these conventions

is necessary because release notes are automatically generated from these messages.

```shell

git commit -a

```

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

- Push your branch to GitHub:

```shell

git push origin my-fix-branch

```

- In GitHub, send a pull request to `RxJS:master`.

- If we suggest changes then:

- Make the required updates.

- Re-run the test suites to ensure tests are still passing.

- Re-run performance tests to make sure your changes didn't hurt performance.

- Rebase your branch and force push to your GitHub repository (this will update your Pull Request):

```shell

git rebase master -i

git push -f

```

- When updating your feature branch with the requested changes, please do not overwrite the commit history, but rather contain the changes in new commits. This is for the sake of a clearer and easier review process.

That's it! Thank you for your contribution!

### 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:

```shell

git push origin --delete my-fix-branch

```

- Check out the master branch:

```shell

git checkout master -f

```

- Delete the local branch:

```shell

git branch -D my-fix-branch

```

- Update your master with the latest upstream version:

```shell

git pull --ff upstream master

```

## Coding Style Guidelines

- Please use proper types and generics throughout your code.

- 2 space indentation only

- favor readability over terseness

(TBD): For now, try to follow the style that exists elsewhere in the source, and use your best judgment.

## Documentation

- The documentation is auto-generated directly from the source code.

- In short: From the source code we generate JSON documents, describing each operator, function ... and render this JSON within an Angular application

- The folder `docs-app` contains everything you need for building and developing the docs

- The [Documentation README](docs\_app/README.md) will support you

- After a PR is merged to master the docs will be published to https://rxjs.dev/

## Unit Tests

Unit tests are located under the [spec directory](/spec). Unit tests over synchronous operators and operations

can be written in a standard [chai](https://www.chaijs.com/) style. Unit tests written against any

asynchronous operator should be written in [Marble Test Style outlined in detail here](doccs\_app/content/guide/testing/internal-marble-tests.md).

Each operator under test must be in its own file to cover the following cases:

- Never

- Empty

- Single/Multiple Values

- Error in the sequence

- Never ending sequences

- Early disposal in sequences

If the operator accepts a function as an argument from the user/developer (for example `filter(fn)` or `zip(a, fn)`),

then it must cover the following cases:

- Success with all values in the callback

- Success with the context, if any allowed in the operator signature

- If an error is thrown

### CI Tests

- Using [Travis](https://travis-ci.org/) on your forked version of RxJS will allow running CI tests on that fork before submitting a PR to master

- Simply create a `Travis` account and add your fork as a new project

- [Sauce Labs](https://saucelabs.com/) setup will allow performing automated browser tests on the fork. Since `saucelabs` doesn't perform browser tests on a PR, this will help verify test results before PR's are checked into master.

- In your `Travis` repo configuration, set the environment variables SAUCE\_USERNAME and SAUCE\_ACCESS\_KEY to your `saucelabs` account ([reference](https://cloud.githubusercontent.com/assets/1210596/12679038/b9ba4eb6-c656-11e5-8c9b-b063c9a3f9dc.png))

- As master runs both of these tests per each check in, it'd be welcome to setup those test before creating your PR

## Performance Tests

One of the primary goals of this library is (and will continue to be) great performance. As such, we've employed a variety of performance

testing techniques.

- DON'T labor over minute variations in ops/sec or milliseconds, there will always be variance in perf test results.

- DON'T alter a performance test unless absolutely necessary. Performance tests may be compared to previous results from previous builds.

- DO run tests multiple times and make sure the margins of error are low

- DO run tests in your feature branches and compare them to master

- DO add performance tests for all new operators

- DO add performance tests that you feel are missing from other operators

- DO add additional performance tests for all worthy code paths. If you develop an operator with special handling for scalar observables,

please add tests for those scenarios

### Macro

[Macro performance tests](perf/macro) are best written for scenarios where many object instance allocations (or deallocations) are occurring. Operators

that create a lot of child subscriptions or operators that emit new objects like Observables and Subjects are definitely worth creating

macro performance tests for.

Other scenarios for macro performance testing may include common end-to-end scenarios from real-world apps. If you have a situation in your

app where you feel RxJS is performing poorly, please [submit an issue](https://github.com/ReactiveX/rxjs/issues/) and include a minimal code example showing

your performance issues. We would love to solve perf for your real-world problems and add those tests to our perf test battery.

Macro performance tests can be run by hosting the root directory with any web server (we use [http-server](https://www.npmjs.com/package/http-server)),

then running:

```sh

npm run build\_all

protractor protractor.conf.js

```

### Micro

[Micro performance tests](perf/micro) really only serve to test operations per second. They're quick and easy to develop, and provide a reasonable look into the

relative performance of our operators versus prior versions. All operators should have corresponding micro performance tests.

Micro performance test can be run with:

```sh

npm run build\_all

node perf/micro

```

If you wish to run a single micro performance test, you can do so by providing a single argument with the name of the perf test file(s):

```sh

node perf/micro zip

```

## Commit Message Guidelines

We have very precise rules over how our git commit messages can be formatted. This leads to \*\*more

readable messages\*\* that are easy to follow when looking through the \*\*project history\*\*. But also,

we use the git commit messages to \*\*generate the RxJS change log\*\*. Helper script `npm run commit`

provides command line based wizard to format commit message easily.

### Commit Message Format

Each commit message consists of a \*\*header\*\*, a \*\*body\*\* and a \*\*footer\*\*. The header has a special

format that includes a \*\*type\*\*, a \*\*scope\*\* and a \*\*subject\*\*:

```

<type>(<scope>): <subject>

<BLANK LINE>

<body>

<BLANK LINE>

<footer>

```

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

Any line of the commit message cannot be longer than 100 characters! This allows the message to be easier

to read on GitHub as well as in various git tools.

### 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.

### Type

Must be one of the following:

- \*\*feat\*\*: A new feature

- \*\*fix\*\*: A bug fix

- \*\*docs\*\*: Documentation only changes

- \*\*style\*\*: Changes that do not affect the meaning of the code (white-space, formatting, missing

semi-colons, etc)

- \*\*refactor\*\*: A code change that neither fixes a bug nor adds a feature

- \*\*perf\*\*: A code change that improves performance

- \*\*test\*\*: Adding missing tests

- \*\*chore\*\*: Changes to the build process or auxiliary tools and libraries such as documentation

generation

### Scope

The scope could be anything specifying the place of the commit change. For example

`Observable`, `Subject`, `switchMap`, etc.

### Subject

The subject contains succinct description of the change:

- use the imperative, present tense: "change" not "changed" nor "changes"

- don't capitalize 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 GitHub issues that this commit \*\*Closes\*\*.

\*\*Breaking Changes\*\* should start with the word `BREAKING CHANGE:` with a space or two newlines. The rest of the commit message is then used for this.