<!-- gen:toc -->

- [How to Contribute](#how-to-contribute)

\* [Contributor License Agreement](#contributor-license-agreement)

\* [Getting Code](#getting-code)

\* [Code reviews](#code-reviews)

\* [Code Style](#code-style)

\* [TypeScript guidelines](#typescript-guidelines)

\* [Project structure and TypeScript compilation](#project-structure-and-typescript-compilation)

- [Shipping CJS and ESM bundles](#shipping-cjs-and-esm-bundles)

- [tsconfig for the tests](#tsconfig-for-the-tests)

- [Root `tsconfig.json`](#root-tsconfigjson)

\* [API guidelines](#api-guidelines)

\* [Commit Messages](#commit-messages)

\* [Writing Documentation](#writing-documentation)

\* [Adding New Dependencies](#adding-new-dependencies)

\* [Running & Writing Tests](#running--writing-tests)

\* [Public API Coverage](#public-api-coverage)

\* [Debugging Puppeteer](#debugging-puppeteer)

- [For Project Maintainers](#for-project-maintainers)

\* [Releasing to npm](#releasing-to-npm)

\* [Updating npm dist tags](#updating-npm-dist-tags)

<!-- gen:stop -->

# How to Contribute

First of all, thank you for your interest in Puppeteer!

We'd love to accept your patches and contributions!

## Contributor License Agreement

Contributions to this project must be accompanied by a Contributor License

Agreement. You (or your employer) retain the copyright to your contribution,

this simply gives us permission to use and redistribute your contributions as

part of the project. Head over to <https://cla.developers.google.com/> to see

your current agreements on file or to sign a new one.

You generally only need to submit a CLA once, so if you've already submitted one

(even if it was for a different project), you probably don't need to do it

again.

## Getting Code

1. Clone this repository

```bash

git clone https://github.com/puppeteer/puppeteer

cd puppeteer

```

2. Install dependencies

```bash

npm install

```

3. Run Puppeteer tests locally. For more information about tests, read [Running & Writing Tests](#running--writing-tests).

```bash

npm run unit

```

## Code reviews

All submissions, including submissions by project members, require review. We

use GitHub pull requests for this purpose. Consult

[GitHub Help](https://help.github.com/articles/about-pull-requests/) for more

information on using pull requests.

## Code Style

- Coding style is fully defined in [`.eslintrc`](https://github.com/puppeteer/puppeteer/blob/main/.eslintrc.js) and we automatically format our code with [Prettier](https://prettier.io).

- It's recommended to set-up Prettier into your editor, or you can run `npm run eslint-fix` to automatically format any files.

- If you're working in a JS file, code should be annotated with [closure annotations](https://github.com/google/closure-compiler/wiki/Annotating-JavaScript-for-the-Closure-Compiler).

- If you're working in a TS file, you should explicitly type all variables and return types. You'll get ESLint warnings if you don't so if you're not sure use them as guidelines, and feel free to ask us for help!

To run ESLint, use:

```bash

npm run eslint

```

You can check your code (both JS & TS) type-checks by running:

```bash

npm run tsc

```

## TypeScript guidelines

- Try to avoid the use of `any` when possible. Consider `unknown` as a better alternative. You are able to use `any` if needbe, but it will generate an ESLint warning.

## Project structure and TypeScript compilation

The code in Puppeteer is split primarily into two folders:

- `src` contains all source code

- `vendor` contains all dependencies that we've vendored into the codebase. See the [`vendor/README.md`](https://github.com/puppeteer/puppeteer/blob/main/vendor/README.md) for details.

We structure these using TypeScript's project references, which lets us treat each folder like a standalone TypeScript project.

### Shipping CJS and ESM bundles

Currently Puppeteer ships two bundles; a CommonJS version for Node and an ESM bundle for the browser. Therefore we maintain two `tsconfig` files for each project; `tsconfig.esm.json` and `tsconfig.cjs.json`. At build time we compile twice, once outputting to CJS and another time to output to ESM.

We compile into the `lib` directory which is what we publish on the npm repository and it's structured like so:

```

lib

- cjs

- puppeteer <== the output of compiling `src/tsconfig.cjs.json`

- vendor <== the output of compiling `vendor/tsconfig.cjs.json`

- esm

- puppeteer <== the output of compiling `src/tsconfig.esm.json`

- vendor <== the output of compiling `vendor/tsconfig.esm.json`

```

The main entry point for the Node module Puppeteer is `cjs-entry.js`. This imports `lib/cjs/puppeteer/index.js` and exposes it to Node users.

### tsconfig for the tests

We also maintain `test/tsconfig.test.json`. This is \*\*only used to compile the unit test `\*.spec.ts` files\*\*. When the tests are run, we first compile Puppeteer as normal before running the unit tests \*\*against the compiled output\*\*. Doing this lets the test run against the compiled code we ship to users so it gives us more confidence in our compiled output being correct.

### Root `tsconfig.json`

The root `tsconfig.json` exists for the API Extractor; it has to find a `tsconfig.json` in the project's root directory. It is \_not\_ used for anything else.

## API guidelines

When authoring new API methods, consider the following:

- Expose as little information as needed. When in doubt, don’t expose new information.

- Methods are used in favor of getters/setters.

- The only exception is namespaces, e.g. `page.keyboard` and `page.coverage`

- All string literals must be small case. This includes event names and option values.

- Avoid adding "sugar" API (API that is trivially implementable in user-space) unless they're \*\*very\*\* demanded.

## Commit Messages

Commit messages should follow the Semantic Commit Messages format:

```

label(namespace): title

description

footer

```

1. \*label\* is one of the following:

- `fix` - puppeteer bug fixes.

- `feat` - puppeteer features.

- `docs` - changes to docs, e.g. `docs(api.md): ..` to change documentation.

- `test` - changes to puppeteer tests infrastructure.

- `style` - puppeteer code style: spaces/alignment/wrapping etc.

- `chore` - build-related work, e.g. doclint changes / travis / appveyor.

2. \*namespace\* is put in parenthesis after label and is optional. Must be lowercase.

3. \*title\* is a brief summary of changes.

4. \*description\* is \*\*optional\*\*, new-line separated from title and is in present tense.

5. \*footer\* is \*\*optional\*\*, new-line separated from \*description\* and contains "fixes" / "references" attribution to github issues.

6. \*footer\* should also include "BREAKING CHANGE" if current API clients will break due to this change. It should explain what changed and how to get the old behavior.

Example:

```

fix(page): fix page.pizza method

This patch fixes page.pizza so that it works with iframes.

Fixes #123, Fixes #234

BREAKING CHANGE: page.pizza now delivers pizza at home by default.

To deliver to a different location, use "deliver" option:

`page.pizza({deliver: 'work'})`.

```

## Writing Documentation

All public API should have a descriptive entry in [`docs/api.md`](https://github.com/puppeteer/puppeteer/blob/main/docs/api.md). There's a [documentation linter](https://github.com/puppeteer/puppeteer/tree/main/utils/doclint) which makes sure documentation is aligned with the codebase.

To run the documentation linter, use:

```bash

npm run doc

```

## Adding New Dependencies

For all dependencies (both installation and development):

- \*\*Do not add\*\* a dependency if the desired functionality is easily implementable.

- If adding a dependency, it should be well-maintained and trustworthy.

A barrier for introducing new installation dependencies is especially high:

- \*\*Do not add\*\* installation dependency unless it's critical to project success.

There are additional considerations for dependencies that are environment agonistic. See the [`vendor/README.md`](https://github.com/puppeteer/puppeteer/blob/main/vendor/README.md) for details.

## Running & Writing Tests

- Every feature should be accompanied by a test.

- Every public api event/method should be accompanied by a test.

- Tests should not depend on external services.

- Tests should work on all three platforms: Mac, Linux and Win. This is especially important for screenshot tests.

Puppeteer tests are located in the test directory ([`test`](https://github.com/puppeteer/puppeteer/blob/main/test/) and are written using Mocha. See [`test/README.md`](https://github.com/puppeteer/puppeteer/blob/main/test/) for more details.

Despite being named 'unit', these are integration tests, making sure public API methods and events work as expected.

- To run all tests:

```bash

npm run unit

```

- To run a specific test, substitute the `it` with `it.only`:

```js

...

it.only('should work', async function({server, page}) {

const response = await page.goto(server.EMPTY\_PAGE);

expect(response.ok).toBe(true);

});

```

- To disable a specific test, substitute the `it` with `xit` (mnemonic rule: '\*cross it\*'):

```js

...

// Using "xit" to skip specific test

xit('should work', async function({server, page}) {

const response = await page.goto(server.EMPTY\_PAGE);

expect(response.ok).toBe(true);

});

```

- To run tests in non-headless mode:

```bash

HEADLESS=false npm run unit

```

- To run Firefox tests, firstly ensure you have Firefox installed locally (you only need to do this once, not on every test run) and then you can run the tests:

```bash

PUPPETEER\_PRODUCT=firefox node install.js

PUPPETEER\_PRODUCT=firefox npm run unit

```

- To run tests with custom browser executable:

```bash

BINARY=<path-to-executable> npm run unit

```

## Public API Coverage

Every public API method or event should be called at least once in tests. To ensure this, there's a `coverage` command which tracks calls to public API and reports back if some methods/events were not called.

Run coverage:

```bash

npm run coverage

```

## Debugging Puppeteer

See [Debugging Tips](README.md#debugging-tips) in the readme.

# For Project Maintainers

## Releasing to npm

Releasing to npm consists of the following phases:

1. Source Code: mark a release.

1. Bump `package.json` version following the SEMVER rules.

2. Run `npm run doc` to update the docs accordingly.

3. Update the “Releases per Chromium Version” list in [`docs/api.md`](https://github.com/puppeteer/puppeteer/blob/main/docs/api.md) to include the new version. Note: only do this when the Chrome revision is different from the previous release.

4. Send a PR titled `'chore: mark version vXXX.YYY.ZZZ'` ([example](https://github.com/puppeteer/puppeteer/pull/5078)).

5. Make sure the PR passes \*\*all checks\*\*.

- \*\*WHY\*\*: there are linters in place that help to avoid unnecessary errors, e.g. [like this](https://github.com/puppeteer/puppeteer/pull/2446)

6. Merge the PR.

7. Once merged, publish the release notes using [GitHub's “draft new release tag” option](https://github.com/puppeteer/puppeteer/releases/new).

- \*\*NOTE\*\*: tag names are prefixed with `'v'`, e.g. for version `1.4.0` the tag is `v1.4.0`.

- For the “raw notes” section, use `git log --pretty="%h - %s" v2.0.0..HEAD`.

2. Publish `puppeteer` to npm.

1. On your local machine, pull from [upstream](https://github.com/puppeteer/puppeteer) and make sure the last commit is the one just merged.

2. Run `git status` and make sure there are no untracked files.

- \*\*WHY\*\*: this is to avoid adding unnecessary files to the npm package.

3. Run [`npx pkgfiles`](https://www.npmjs.com/package/pkgfiles) to make sure you don't publish anything unnecessary.

4. Run `npm publish`. This publishes the `puppeteer` package.

3. Publish `puppeteer-core` to npm.

1. Run `./utils/prepare\_puppeteer\_core.js`. The script changes the name inside `package.json` to `puppeteer-core`.

2. Run `npm publish`. This publishes the `puppeteer-core` package.

3. Run `git reset --hard` to reset the changes to `package.json`.

4. Source Code: mark post-release.

1. Bump `package.json` version to `-post` version, run `npm run doc` to update the “released APIs” section at the top of `docs/api.md` accordingly, and send a PR titled `'chore: bump version to vXXX.YYY.ZZZ-post'` ([example](https://github.com/puppeteer/puppeteer/commit/d02440d1eac98028e29f4e1cf55413062a259156))

- \*\*NOTE\*\*: no other commits should be landed in-between release commit and bump commit.

## Updating npm dist tags

For both `puppeteer` and `puppeteer-core` we maintain `chrome-\*` npm dist tags, e.g. `chrome-75` and so on. These tags match the Puppeteer version that corresponds to the `chrome-\*` release.

These tags are updated on every Puppeteer release.

Managing tags 101:

```bash

# List tags

$ npm dist-tag ls puppeteer

# Add tags

$ npm dist-tag add puppeteer@3.0.0 chrome-81

$ npm dist-tag add puppeteer-core@3.0.0 chrome-81

```