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

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

2. Install dependencies

```
npm install
```

3. Run Puppeteer tests locally. For more information about tests, read Running & Writing Tests.

```
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 for more information on using pull requests.

Code Style

- Coding style is fully defined in `.eslintrc` and we automatically format our code with Prettier.
- 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.
- 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:

```
npm run eslint
```

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

```
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` 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 Conventional Commits format. This is enforced via `npm run lint`.

In particular, breaking changes should clearly be noted as "BREAKING CHANGE:" in the commit message footer. Example:

```
fix(page): fix page.pizza method

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

Issues: #123, #234

BREAKING CHANGE: page.pizza now delivers pizza at home by default.
To deliver to a different location, use the "deliver" option:
  `page.pizza({deliver: 'work'})`.
```

Writing Documentation

All public API should have a descriptive entry in `docs/api.md`. There's a documentation linter which makes sure documentation is aligned with the codebase.

To run the documentation linter, use:

```
npm run doc
```

To format the documentation markdown and its code snippets, use:

```
npm run prettier-fix
```

Writing TSDoc Comments

Each change to Puppeteer should be thoroughly documented using TSDoc comments. Refer to the API Extractor documentation for information on the exact syntax.

- Every new method needs to have either `@public` or `@internal` added as a tag depending on if it is part of the public API.
- Keep each line in a comment to no more than 90 characters (ESLint will warn you if you go over this). If you're a VSCode user the Rewrap plugin is highly recommended!

Running New Documentation website locally

- In the Puppeteer's folder, install all dependencies with `npm i`.

- run `npm run generate-docs` which will generate all the `.md` files on `puppeteer/website/docs`.
- run `npm i` on `puppeteer/website`.
- run `npm start` on `puppeteer/website`.

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` 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 and are written using Mocha. See `test/README.md` 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:

```
npm run unit
```

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

```
...
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*'):

```
...
// 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:

```
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:

```
PUPPETEER_PRODUCT=firefox node install.js
PUPPETEER_PRODUCT=firefox npm run unit
```

- To run tests with custom browser executable:

```
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:

```
npm run coverage
```

Debugging Puppeteer

See Debugging Tips in the readme.

For Project Maintainers

Rolling new Chromium version

The following steps are needed to update the Chromium version.

1. Find a suitable Chromium revision Not all revisions have builds for all platforms, so we need to find one that does. To do so, run `utils/check_availability.js -rd` to find the latest suitable `dev` Chromium revision (see `utils/check_availability.js -help` for more options).
2. Update `src/revisions.ts` with the found revision number.
3. Update `versions.js` with the new Chromium-to-Puppeteer version mapping and update `lastMaintainedChromiumVersion` with the latest stable Chrome version.

4. Run `npm run ensure-correct-devtools-protocol-revision`. If it fails, update `package.json` with the expected `devtools-protocol` version.
5. Run `npm run tsc` and `npm install`.
6. Run `npm run unit` and ensure that all tests pass. If a test fails, bisect the upstream cause of the failure, and either update the test expectations accordingly (if it was an intended change) or work around the changes in Puppeteer (if it's not desirable to change Puppeteer's observable behavior).
7. Commit and push your changes and open a pull request. The commit message must contain the version in `Chromium <version> (<revision>)` format to ensure that pptr.dev can parse it correctly, e.g. `'feat(chromium): roll to Chromium 90.0.4427.0 (r856583)'`.

Bisecting upstream changes

Sometimes, performing a Chromium roll causes tests to fail. To figure out the cause, you need to bisect Chromium revisions to figure out the earliest possible revision that changed the behavior. The script in `utils/bisect.js` can be helpful here. Given a pattern for one or more unit tests, it will automatically bisect the current range:

```
node utils/bisect.js --good 686378 --bad 706915 script.js

node utils/bisect.js --unit-test Response.fromCache
```

By default, it will use the Chromium revision in `src/revisions.ts` from the `main` branch and from the working tree to determine the range to bisect.

Releasing to npm

Releasing to npm consists of the following phases:

1. Source Code: mark a release.
    1. Run `npm run release`. (This automatically bumps the version number in `package.json`, populates the changelog, updates the docs, and creates a Git commit for the next step.)
    2. Send a PR for the commit created in the previous step.
    3. Make sure the PR passes **all checks**.
        - **WHY**: there are linters in place that help to avoid unnecessary errors, e.g. like this
    4. Merge the PR.
    5. Once merged, publish the release notes from `CHANGELOG.md` using GitHub's "draft new release tag" option.
        - **NOTE**: tag names are prefixed with `'v'`, e.g. for version `1.4.0` the tag is `v1.4.0`.
    6. As soon as the Git tag is created by completing the previous step, our CI automatically `npm publish`es the new releases for both the `puppeteer` and `puppeteer-core` packages.

2. Source Code: mark post-release.
    1. Bump `package.json` version to the `-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)
        - **NOTE**: no other commits should be landed in-between release commit and bump commit.