# Contributing

This document is intended for developers interest in making contributions to Preact and document our internal processes like releasing a new version.

## Getting Started

This steps will help you to set up your development environment. That includes all dependencies we use to build Preact and developer tooling like git commit hooks.

1. Clone the git repository: `git clone git@github.com:preactjs/preact.git`

2. Go into the cloned folder: `cd preact/`

3. Install all dependencies: `npm install`

## The Repo Structure

This repository contains Preact itself, as well as several addons like the debugging package for example. This is reflected in the directory structure of this repository. Each package has a `src/` folder where the source code can be found, a `test` folder for all sorts of tests that check if the code in `src/` is correct, and a `dist/` folder where you can find the bundled artifacts. Note that the `dist/` folder may not be present initially. It will be created as soon as you run any of the build scripts inside `package.json`. More on that later ;)

A quick overview of our repository:

```bash

# The repo root (folder where you cloned the repo into)

/

src/ # Source code of our core

test/ # Unit tests for core

dist/ # Build artifacts for publishing on npm (may not be present)

# Sub-package, can be imported via `preact/compat` by users.

# Compat stands for react-compatibility layer which tries to mirror the

# react API as close as possible (mostly legacy APIs)

compat/

src/ # Source code of the compat addon

test/ # Tests related to the compat addon

dist/ # Build artifacts for publishing on npm (may not be present)

# Sub-package, can be imported via `preact/hooks` by users.

# The hooks API is an effect based API to deal with component lifcycles.

# It's similar to hooks in React

hooks/

src/ # Source code of the hooks addon

test/ # Tests related to the hooks addon

dist/ # Build artifacts for publishing on npm (may not be present)

# Sub-package, can be imported via `preact/debug` by users.

# Includes debugging warnings and error messages for common mistakes found

# in Preact application. Also hosts the devtools bridge

debug/

src/ # Source code of the debug addon

test/ # Tests related to the debug addon

dist/ # Build artifacts for publishing on npm (may not be present)

# Sub-package, can be imported via `preact/test-utils` by users.

# Provides helpers to make testing Preact applications easier

test-utils/

src/ # Source code of the test-utils addon

test/ # Tests related to the test-utils addon

dist/ # Build artifacts for publishing on npm (may not be present)

# A demo application that we use to debug tricky errors and play with new

# features.

demo/

# Contains build scripts and dependencies for development

package.json

```

\_Note: The code for rendering Preact on the server lives in another repo and is a completely separate npm package. It can be found here: [https://github.com/preactjs/preact-render-to-string](https://github.com/preactjs/preact-render-to-string)\_

### What does `mangle.json` do?

It's a special file that can be used to specify how `terser` (previously known as `uglify`) will minify variable names. Because each sub-package has it's own distribution files we need to ensure that the variable names stay consistent across bundles.

## What does `options.js` do?

Unique to Preact we do support several ways to hook into our renderer. All our addons use that to inject code at different stages of a render process. They are documented in our typings in `internal.d.ts`. The core itself doesn't make use of them, which is why the file only contains an empty `object`.

## Important Branches

We merge every PR into the `master` branch which is the one that we'll use to publish code to npm. For the previous Preact release line we have a branch called `8` which is in maintenance mode. As a new contributor you won't have to deal with that ;)

## Creating your first Pull-Request

We try to make it as easy as possible to contribute to Preact and make heavy use of GitHub's "Draft PR" feature which tags Pull-Requests (short = PR) as work in progress. PRs tend to be published as soon as there is an idea that the developer deems worthwhile to include into Preact and has written some rough code. The PR doesn't have to be perfect or anything really ;)

Once a PR or a Draft PR has been created our community typically joins the discussion about the proposed change. Sometimes that includes ideas for test cases or even different ways to go about implementing a feature. Often this also includes ideas on how to make the code smaller. We usually refer to the latter as "code-golfing" or just "golfing".

When everything is good to go someone will approve the PR and the changes will be merged into the `master` branch and we usually cut a release a few days/ a week later.

\_The big takeaway for you here is, that we will guide you along the way. We're here to help to make a PR ready for approval!\_

The short summary is:

1. Make changes and submit a PR

2. Modify change according to feedback (if there is any)

3. PR will be merged into `master`

4. A new release will be cut (every 2-3 weeks).

## Commonly used scripts for contributions

Scripts can be executed via `npm run [script]` or `yarn [script]` respectively.

- `build` - compiles all packages ready for publishing to npm

- `build:core` - builds just Preact itself

- `build:debug` - builds the debug addon only

- `build:hooks` - builds the hook addon only

- `build:test-utils` - builds the test-utils addon only

- `test:ts` - Run all tests for TypeScript definitions

- `test:karma` - Run all unit/integration tests.

- `test:karma:watch` - Same as above, but it will automatically re-run the test suite if a code change was detected.

But to be fair, the only ones we use ourselves are `build` and `test:karma:watch`. The other ones are mainly used on our CI pipeline and we rarely use them.

\_Note: Both `test:karma` and `test:karma:watch` listen to the environment variable `COVERAGE=true`. Disabling code coverage can significantly speed up the time it takes to complete the test suite.\_

\_Note2: The test suite is based on `karma` and `mocha`. Individual tests can be executed by appending `.only`:\_

```jsx

it.only('should test something', () => {

expect(1).to.equal(1);

});

```

## Common terminology and variable names

- `vnode` -> shorthand for `virtual-node` which is an object that specifies how a Component or DOM-node looks like

- `commit` -> A commit is the moment in time when you flush all changes to the DOM

- `c` -> The variable `c` always refers to a `component` instance throughout our code base.

- `diff/diffing` -> Diffing describes the process of comparing two "things". In our case we compare the previous `vnode` tree with the new one and apply the delta to the DOM.

- `root` -> The topmost node of a `vnode` tree

## Tips for getting to know the code base

- Check the JSDoc block right above the function definition to understand what it does. It contains a short description of each function argument and what it does.

- Check the callsites of a function to understand how it's used. Modern editors/IDEs allow you to quickly find those, or use the plain old search feature instead.

## FAQ

### Why does the JSDoc use TypeScript syntax to specify types?

Several members of the team are very fond of TypeScript and we wanted to leverage as many of its advantages, like improved autocompletion, for Preact. We even attempted to port Preact to TypeScript a few times, but we ran into many issues with the DOM typings. Those would force us to fill our codebase with many `any` castings, making our code very noisy.

Luckily TypeScript has a mode where it can somewhat reliably typecheck JavaScript code by reusing the types defined in JSDoc blocks. It's not perfect and it often has trouble inferring the correct types the further one strays away from the function arguments, but it's good enough that it helps us a lot with autocompletion. Another plus is that we can make sure that our TypeScript definitons are correct at the same time.

Check out the [official TypeScript documentation](https://www.typescriptlang.org/docs/handbook/type-checking-javascript-files.html) for more information.

\_Note that we have separate tests for our TypeScript definition files. We only use `ts-check` for local development and don't check it anywhere else like on the CI.\_

### Why does the code base often use `let` instead of `const`?

There is no real reason for that other a historical one. Back before auto-formatting via prettier was a thing and minifiers weren't as advanced as they are today we used a pretty terse code-style. The code-style deliberately was aimed at making code look as concise and short as possible. The `let` keyword is a bit shorter than `const` to write, so we only used that. This was done only for stylistic reasons.

This helped our minds to not lose sight of focusing on size, but made it difficult for newcomers to start contributing to Preact. For that reason alone we switched to `prettier` and loosened our rule regarding usage of `let` or `const`. Today we use both, but you can still find many existing places where `let` is still in use.

In the end there is no effect on size regardless if you use `const`, `let` or use both. Our code is downtranspiled to `ES5` for npm so both will be replaced with `var` anyways. Therefore it doesn't really matter at all which one is used in our codebase.

This will only become important once shipping modern JavaScript code on npm becomes a thing and bundlers follow suit.

## How to create a good bug report

To be able to fix issues we need to see them on our machine. This is only possible when we can reproduce the error. The easiest way to do that is narrow down the problem to specific components or combination of them. This can be done by removing as much unrelated code as possible.

The perfect way to do that is to make a [codesandbox](https://codesandbox.io/). That way you can easily share the problematic code and ensure that others can see the same issue you are seeing.

For us a [codesandbox](https://codesandbox.io/) says more than a 1000 words :tada:

## I have more questions on how to contribute to Preact. How can I reach you?

We closely watch our issues and have a pretty active [Slack workspace](https://preact-slack.now.sh/). Nearly all our communication happens via these two forms of communication.

## Releasing Preact (Maintainers only)

This guide is intended for core team members that have the necessary

rights to publish new releases on npm.

1. [Write the release notes](#writing-release-notes) and keep them as a draft in GitHub

1. I'd recommend writing them in an offline editor because each edit to a draft will change the URL in GitHub.

2. Make a PR where \*\*only\*\* the version number is incremented in `package.json` (note: We follow `SemVer` conventions)

3. Wait until the PR is approved and merged.

4. Switch back to the `master` branch and pull the merged PR

5. Run `npm run build && npm publish`

1. Make sure you have 2FA enabled in npm, otherwise the above command will fail.

2. If you're doing a pre-release add `--tag next` to the `npm publish` command to publish it under a different tag (default is `latest`)

6. Publish the release notes and create the correct git tag.

7. Tweet it out

## Legacy Releases (8.x)

> \*\*ATTENTION:\*\* Make sure that you've cleared the project correctly

> when switching from a 10.x branch.

0. Run `rm -rf dist node\_modules && npm i` to make sure to have the correct dependencies.

Apart from that it's the same as above.

## Writing release notes

The release notes have become a sort of tiny blog post about what's

happening in preact-land. The title usually has this format:

```txt

Version Name

```

Example:

```txt

10.0.0-beta.1 Los Compresseros

```

The name is optional, we just have fun finding creative names :wink:

To keep them interesting we try to be as

concise as possible and to just reflect where we are. There are some

rules we follow while writing them:

- Be nice, use a positive tone. Avoid negative words

- Show, don't just tell.

- Be honest.

- Don't write too much, keep it simple and short.

- Avoid making promises and don't overpromise. That leads to unhappy users

- Avoid framework comparisons if possible

- Highlight awesome community contributions (or great issue reports)

- If in doubt, praise the users.

After this section we typically follow with a changelog part that's

divided into 4 groups in order of importance for the user:

- Features

- Bug Fixes

- Typings

- Maintenance

We generate it via this handy cli program: [changelogged](https://github.com/marvinhagemeister/changelogged). It will collect and format

the descriptions of all PRs that have been merged between two tags.

The usual command is `changelogged 10.0.0-rc.2..HEAD` similar to how

you'd diff two points in time with git. This will get you 90% there,

but you still need to divide it into groups. It's also a good idea

to unify the formatting of the descriptions, so that they're easier

to read and don't look like a mess.