# Developing the Garden CLI framework

## Contributing guidelines

We heartily welcome any form of contribution to the project, including issue reports, feature requests,

discussion, pull requests and any type of feedback. We request that all contributors

adhere to the [Contributor Covenant](CODE\_OF\_CONDUCT.md) and work with us to make the collaboration and

community productive and fun for everyone :)

## Setting up your development environment

### Step 1: Install Docker and Kubernetes

Please refer to our [installation docs](./docs/basics/installation.md) for instructions on how to install Docker and Kubernetes for different platforms.

### Step 2: Clone the repo

git clone https://github.com/garden-io/garden.git

### Step 3: Install dependencies

#### OSX

For Mac we have a script that installs all required dependencies.

If you haven't already, please [install Homebrew](https://docs.brew.sh/Installation). Then run:

./bin/bootstrap-osx

#### Windows / Linux

Other platforms need to roll their own for now (contributions welcome!). Please have a look at the script for OSX to see what's installed.

\*\*Note:\*\* We recommend using Node 12 when developing Garden.

### Step 4: Bootstrap project

Install Node modules for the root package, and the `dashboard` and `garden-service` packages:

```sh

npm install # To install root dependencies

npm run bootstrap # To bootstrap packages

```

from the root directory

You may need to install the Node modules in the garden-service package manually due to [lerna/lerna#1457](https://github.com/lerna/lerna/issues/1457).

```sh

cd garden-service

npm install

```

## Developing Garden

### Initial build

Before running Garden for the first time, you need to do an initial build by running

```sh

npm run build

```

from the root directory. This ensures that the dashboard is built and ready to serve and that version files are in place.

### Developing

To develop the CLI, run the `dev` command in your console:

cd garden-service

npm run dev

This will `npm link` it to your global npm folder, and then watch for

changes and auto-rebuild as you code. You can then run the `garden` command as normal.

Also, you might like to add a couple of shorthands:

```sh

alias g='garden'

alias k='kubectl'

```

For developing the dashboard, please refer to the [dashboard docs](./dashboard/README.mdj).

### Formatting

We use [Prettier](https://prettier.io) for automated formatting. We highly recommend installing the appropriate plugin for your editor to automate formatting as you work on your code. You can also run `npm run fix-format` to fix formatting across the codebase.

### Debugging

To enable setting a breakpoint in the code, run the CLI with the `garden-service/bin/static/garden-debug` binary, which adds the `--inspect` flag. Developers might find it useful to alias this:

```sh

alias gd='/path/to/garden/bin/garden-debug'

```

You can e.g. use the Chrome DevTools to inspect the code at the breakpoint:

1. Add a `debugger` statement somewhere in the code.

2. Navigate to [chrome://inspect/](chrome://inspect/) in your Chrome browser.

3. Click the \*\*Open dedicated DevTools for Node\*\* link.

4. Run a CLI command that hits the breakpoint, e.g.:

```sh

/path/to/garden/bin/garden-debug deploy # or gd deploy, if you've set the alias

```

You should now be able to inspect the code at run time in the \*\*Console\*\* tab of the DevTools window.

### Tests

Unit tests are run using `mocha` via `npm test` from the directory of the package you want to test. To run a specific test, you can grep the test description with the `-g` flag.:

```sh

cd garden-service

npm test # run all unit tests

npm test -- -g "taskGraph" # run only tests with descriptions matching "taskGraph"

```

Integration tests are run with:

```sh

npm run integ

```

End-to-end tests are run with:

```sh

npm run e2e

```

You can also run the end-to-end tests for a specific example project using:

```sh

npm run e2e-project -- --project=<example project name>

```

End to end tests are run in CI by using Garden itself to test the project defined in `./garden-service/test/e2e/garden.yml`. Cf. the appropriate job in `circleci/config.yml` for details.

### Commit messages

We follow the [Conventional Commits specification](https://www.conventionalcommits.org/en/v1.0.0-beta.4/#specification) and automatically validate the formatting of our commit messages. In particular, the \*\*type\*\* of the commit header must be one of the following:

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

generation.

\* \*\*ci\*\*: Changes to the CI configuration.

\* \*\*docs\*\*: Documentation only changes.

\* \*\*feat\*\*: A new feature.

\* \*\*fix\*\*: A bug fix.

\* \*\*improvement\*\*: Changes that improve a current implementation without adding a new feature or fixing a bug.

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

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

\* \*\*revert\*\*: A commit that 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.

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

semi-colons, etc).

\* \*\*test\*\*: Adding missing or correcting existing tests.

When generating the changelog, we only include the following types: \*\*feat\*\*, \*\*fix\*\*, \*\*refactor\*\*, \*\*improvement\*\*, and \*\*perf\*\*. This means that any changes that the user should be aware of, should have one of these types.

### Checks

We have scripts for checking licenses, docs, linting and more. These can all be run with a single command:

```sh

npm run check-all

```

If the checks fail because of bad formatting, run:

```sh

npm run fix-format

```

If the checks fail because of missing docs, run:

```sh

npm run generate-docs

```

### Pre-push hook

Before pushing, we automatically run the `check-all` script from above, as well as unit tests. To skip these, run push with the `--no-verify` flag:

```sh

git push origin <my-branch> --no-verify

```

### Environment Variables

You should set the following environment variables when developing on Garden:

```sh

GARDEN\_DISABLE\_ANALYTICS=true

GARDEN\_DISABLE\_VERSION\_CHECK=true

ANALYTICS\_DEV=true

```

## CI

We use [Circle CI](https://circleci.com) for integration and end to end testing. Sometimes

it can be useful to test and debug the CI build locally, particularly when

updating or adding dependencies. You can use their

[CLI](https://circleci.com/docs/2.0/local-jobs/) for that, which

is installed automatically by the `./bin/bootstrap-osx` script. Once you

have it installed you can run `circleci build` in the repo root to test

the build locally.

## License/copyright headers

Every source file must include the contents of `static/license-header.txt` at the top. This is

automatically checked during CI. You can run the check with `npm run check-licenses`.

## Release process

### Packages

Our release process generates the following packages:

\* An executable for OSX, Linux, and Windows, generated by [Pkg](https://github.com/zeit/pkg) and hosted on our [Github page](https://github.com/garden-io/garden/releases).

\* A [Homebrew](https://brew.sh/) package for OSX users.

### Process

We have a dedicated release branch, `latest-release`, off of which we create our releases using our [release script](https://github.com/garden-io/garden/blob/master/bin/release.ts). Once we're ready to release, we reset the `latest-release` branch to `master` and create a pre-release with the script. If there are issues with the pre-release, we merge the fixes to `master` and cherry pick them to the `latest-release` branch. We repeat this process until all issues have been resolved and we can make a proper release.

This procedure allows us to continue merging features into `master` without them being included in the release.

On every merge to `master` we also publish an \*\*unstable\*\* release with the version `edge` that is always flagged as a pre-release.

### Release script

The [release script](https://github.com/garden-io/garden/blob/master/bin/release.ts) has the signature:

```sh

./bin/release.tsx <minor | patch | preminor | prepatch | prerelease> [--force] [--dry-run]

```

and does the following:

\* Checks out a branch named `release-<version>`.

\* Updates `garden-service/package.json`, `garden-service/package-lock.json` and `CHANGELOG.md`.

\* Commits the changes, tags the commit, and pushes the tag and branch.

\* Pushing the tag triggers a CI process that creates the release artifacts and publishes them to Github. If the the release is not a pre-release, we create a draft instead of actually publishing.

### Steps

To make a new release, set your current working directory to the garden root directory and follow the steps below.

1. \*\*Checkout to the `latest-release` branch\*\*.

2. The next step depends on the release type:

\* If you're making the first pre-release:

1. Reset `latest-release` to `master` with `git reset --hard origin/master`.

2. Run `./bin/release.ts preminor|prepatch`.

\* If youve already created a pre-release, e.g. `v1.2.3-alpha.0`, and want to create a new pre-release `v1.2.3-alpha.1` which includes fixes merged to master since `v1.2.3-alpha.0` was created, do the following:

1. Checkout to the most recent pre-release branch, in this case `v1.2.3-alpha.0`, and cherry-pick the appropriate commits from `master`.

2. Run `./bin/release.ts prerelease`.

\* If youre ready to make a proper release, do the following:

1. Checkout to the most recent pre-release branch, e.g. `v1.2.3-alpha.1`.

2. Remove all the `bump version...` commits. E.g. by using `git rebase -i <hash-before-first-version-bump>` and `drop`-ing the commits. In this case we drop `chore(release): bump version to v1.2.3-alpha.0` and `chore(release): bump version to v.1.2.3-alpha.1`.

3. Run `./bin/release.ts minor | patch`. This way, the version bump commits and changelog entries created by the pre-releases are omitted from the final history.

3. If you're making a pre-release you're done, and you can now start testing the binaries that were just published to our Github [Releases page](https://github.com/garden-io/garden/releases) (\*\*step 4\*\*). Otherwise go to \*\*step 5\*\*.

4. Manual testing (using the pre-release/release binary)

\* On a \*\*Windows\*\* machine, run `garden dev --hot=vote` in the `vote` example project.

\* Change a file in the `vote` service and verify that the hot reload was successful.

\* Open the dashboard, verify that the initial page loads without errors.

\* On \*\*macOS\*\* or \*\*Linux\*\*, run the `./bin/test-release <version>` script. The script runs some simple tests to sanity check the release.

5. Go to our Github [Releases page](https://github.com/garden-io/garden/releases) and click the \*\*Edit\*\* button for the draft just created from CI. Note that for drafts, a new one is always created instead of replacing a previous one.

6. Write release notes. The notes should give an overview of the release and mention all relevant features. They should also \*\*acknowledge all external contributors\*\* and contain the changelog for that release.

\* To generate a changelog for just that tag, run `git-chglog <tag-name>`

\* To get a list of all contributors between releases, ordered by count, run: `git log <previous-tag>..<current-tag> --no-merges "$@" | grep ^Author | sort | uniq -c | sort -nr`. Note that authors of squashed commits won't show up so it might be good to do a quick sanity check on Github as well.

7. Click the \*\*Publish release\*\* button.

8. Make a pull request for the branch that was pushed by the script.

9. Make sure the `latest-release` branch contains the released version, and push it to the remote. \*\*This branch is used for our documentation, so this step is important.\*\*

10. Check the `update-homebrew` GitHub Action run succesfully and the [homebrew repo](https://github.com/garden-io/homebrew-garden) contains the latest version.

11. Install the Homebrew package and make sure it works okay:

\* `brew tap garden-io/garden && brew install garden-cli || true && brew update && brew upgrade garden-cli`

\* Run `$(brew --prefix garden-cli)/bin/garden dev` (to make sure you're using the packaged release) in an example project and see if all looks well.

12. Prepare the release announcement and publish it in our channels (Slack and Twitter). If not possible, delegate the task to an available contributor.

## Changelog

We keep a changelog under `CHANGELOG.md` that get's updated on every release. For pre-releases, we include every pre-release tag in that release cycle in the changelog. So if we're releasing, say, `v0.9.1-3`, the changelog will include entries for `v0.9.1-0`, `v0.9.1-1`, `v0.9.1-2`, assuming those tags exist. Once we make a proper release, we remove the pre-release tags so that the changelog only shows changes between `v0.9.0` and `v0.9.1`. A changelog with the pre-releases is of course always available in our Git history.