# CONTRIBUTING

Contributions are always welcome, no matter how large or small. Before contributing,

please read the [code of conduct](CODE\_OF\_CONDUCT.md).

## Setup

### installation

- install dependencies using npm

```bash

$ npm i

```

- install command globally and link (in order to use the "bit-dev" command globally and always use the

latest development build)

```bash

npm run dev-link

```

if you want your command to be different then the default (bit-dev) just add your favorite command name as an argument to the script

```bash

npm run dev-link my-bit-dev-cmd-name

```

for example:

```bash

npm run dev-link bit-dev

```

bit will install these commands in `/usr/local/bin/` directory, so in order to remove them just use the bash `rm` command.

```bash

rm /usr/local/bin/my-bit-dev-cmd-name

```

### Build

- build distributions:

```bash

npm run build

```

- use with watch, to run the build on every code modification

```bash

npm run watch

```

### Unit Tests

- run the unit tests

```bash

npm test

```

### End to End Tests

- run the e2e tests (with default 'bit' command)

```bash

npm run e2e-test

```

- run e2e tests using bit-dev command

```bash

npm run e2e-test --bit\_bin=bit-dev

```

- run e2e-tests for debugging (shows output and doesn't delete the workspaces)

```bash

npm run e2e-test:debug

```

- run e2e-tests for SSH (switch from exporting by using file-system to SSH approach). Make sure you are able to run 'ssh `whoami`@127.0.0.1' on your local.

```bash

npm run e2e-test:ssh

npm run e2e-test:ssh-debug

```

When adding end to end tests please make sure new test files are created in the following name convention: ```name.e2e.[number].js``` where number should be 1/2/3. This was made in order to batch work in appveyor.

Keep in mind that running the e2e-tests locally may take hours to complete, it's faster to create a new PR and let CircleCI run them. Circle is configured to run multiple tests in parallel and complete them much faster.

### Debugging

The code is heavy on promises, as such, some errors don't have a useful stack trace. Bluebird enables the long stack trace when the env is development or when `BLUEBIRD\_DEBUG` is set. Normally, the full stack trace is not shown on the console but logged in the debug.log file. (located at /Users/your-use/Library/Caches/Bit/logs/debug.log on Mac).

In some cases, you might get very helpful info by prefixing Bit command with `BIT\_LOG=\*`. For now, this helps to get more info about why a component is shown as modified and it also shows the events for `bit watch`.

To print the log messages on the console, prefix your command with `BIT\_LOG=<debug-level>`, e.g. `BIT\_LOG=error`.

The log level written to the log file is by default "debug". To change it, run `bit config set log\_level <level>`, e.g. `bit config set log\_level silly`.

### Lint

Run eslint and tsc (for type checking)

```bash

npm run lint

```

## Pull Requests

We actively welcome your pull requests.

1. Fork the repo and create your branch from `master`.

2. If you've added code that should be tested, add tests.

3. Ensure the test suite passes.

4. Make sure your code lints.

5. Add your change to the CHANGELOG.md file at the [unreleased] section.

## Understanding the code

A standard flow is as follows:

1. A user enters Bit command.

2. We use `commander` package to help with parsing the command. A command file in `src/cli/commands` takes care of the specific command details.

3. A command in `cli/commands/<command-name>` goes to the API, which is in `src/api` and sends the action to run

4. The API has two main parts: Consumer and Scope, which are the two main concepts in Bit source code. See below explanations about them.

In general, if a command related to the user local components it goes to the consumer, otherwise, it goes to the scope.

5. The API loads the Consumer or the Scope and pass them the action to run.

6. The command receives the response from the API as Promise and prints the results to the user.

### Consumer

The user workspace. The consumer root is the user workspace root directory.

It has a .bitmap file where it stores all the components locations on the filesystem.

It also has a bit.json file where it stores the user configuration, such as what compilers and testers are used.

When the user adds a new component (using `bit add` command), it is done on the consumer part only. The files are added to the .bitmap file and it doesn't involve the scope.

### Scope

The model / objects. The scope root is located at `<consumer-root>.bit` or `<consumer-root>/.git/.bit`.

When a new component is tagged (using `bit tag` command), Bit compresses the component files and save them along with metadata about the component in the scope.

The file names are the hashes of the files/metadata. Think about it as .git in Git world.

The main players in the scope are: Component, Version and Source. All of them inherit from BitObject class.

`Component` has the general data about the component, such as the name, the scope and the versions.

`Version` has the data about a particular version of the component, it stores (among other things) the hashes of the files of that version.

`Source` is the actual file. It can be the component file or the dist file generated by Bit compiler.

## License

By contributing to Bit, you agree that your contributions will be licensed

under its [Apache2 license](LICENSE).