---

<p align="center" class="toc">

<strong><a href="#setup">Setup</a></strong>

|

<strong><a href="#running-lintingtests">Running linting/tests</a></strong>

|

<strong><a href="#writing-tests">Writing tests</a></strong>

|

<strong><a href="#debugging-code">Debugging code</a></strong>

|

<strong><a href="#internals">Internals</a></strong>

</p>

---

# Contributing

Contributions are always welcome, no matter how large or small! Before contributing, please read the [code of conduct](https://github.com/babel/babel/blob/main/CODE\_OF\_CONDUCT.md).

If you want an already configured online IDE to contribute to Babel, you can use [Gitpod](https://gitpod.io/#https://github.com/babel/babel)!

## Not sure where to start?

- If you aren't just making a documentation change, you'll probably want to learn a bit about a few topics.

- [ASTs](https://en.wikipedia.org/wiki/Abstract\_syntax\_tree) (Abstract Syntax Tree): The Babel AST [spec](https://github.com/babel/babel/blob/main/packages/babel-parser/ast/spec.md) is a bit different from [ESTree](https://github.com/estree/estree). The differences are listed [here](https://babeljs.io/docs/en/next/babel-parser.html#output).

- Check out [`/doc`](https://github.com/babel/babel/tree/main/doc) for information about Babel's internals

- Check out [the Babel Plugin Handbook](https://github.com/thejameskyle/babel-handbook/blob/master/translations/en/plugin-handbook.md#babel-plugin-handbook) - core plugins are written the same way as any other plugin!

- Check out [AST Explorer](http://astexplorer.net/#/scUfOmVOG5) to learn more about ASTs or make your own plugin in the browser

- When you feel ready to jump into the Babel source code, a good place to start is to look for issues tagged with [help wanted](https://github.com/babel/babel/labels/help%20wanted) and/or [good first issue](https://github.com/babel/babel/labels/good%20first%20issue).

- Follow along with what we are working on by joining our [Slack](https://babeljs.slack.com) (you can sign-up [here](https://slack.babeljs.io/)

for an invite), following our announcements on [Twitter](https://twitter.com/babeljs), and reading (or participating!) in our [meeting notes](https://github.com/babel/notes).

- Check out our [website](http://babeljs.io/) and the [repo](https://github.com/babel/website)

- You can contribute by triaging issues which may include reproducing bug reports or asking for vital information, such as version numbers or reproduction instructions. If you would like to start triaging issues, one easy way to get started is to [subscribe to babel on CodeTriage](https://www.codetriage.com/babel/babel). [![Open Source Helpers](https://www.codetriage.com/babel/babel/badges/users.svg)](https://www.codetriage.com/babel/babel)

## Chat

Feel free to check out the `#discussion`/`#development` channels on our [Slack](https://slack.babeljs.io) (you can sign-up [here](https://slack.babeljs.io/) for an invite). Some of us are always online to chat!

## Developing

\*Node\*: Check that Node is [installed](https://nodejs.org/en/download/) with version 10 and up. You can check this with `node -v`.

\*Yarn\*: Make sure that Yarn 1 is [installed](https://classic.yarnpkg.com/en/docs/install) with version >= `1.19.0`.

\*Make\*: If you are running Windows 10, you'll need to do one of the following:

\* Clone the repository and run the commands inside [WSL 2](https://docs.microsoft.com/en-us/windows/wsl/install-win10).

\* Install [Make for Windows](http://gnuwin32.sourceforge.net/packages/make.htm).

### Setup

Fork the `babel` repository to your GitHub Account.

Then, run:

```sh

$ git clone https://github.com/<your-github-username>/babel

$ cd babel

$ make bootstrap

```

Then you can either run:

```sh

$ make build

```

to build Babel \*\*once\*\* or:

```sh

$ make watch

```

to have Babel build itself and incrementally build files on change.

> You can access the built files for individual packages from `packages/<package-name>/lib`.

If you wish to build a copy of Babel for distribution, then run:

```sh

$ make build-dist

```

### Running linting/tests

You can run lint via:

```sh

# ~6 sec on a MacBook Pro (Mid 2015)

$ make lint

```

You can run eslint's autofix via:

```sh

$ make fix

```

You can run tests + lint for all packages (slow) via:

```sh

# ~46 sec on a MacBook Pro (Mid 2015)

$ make test

```

If you just want to run all tests:

```sh

# ~40 sec on a MacBook Pro (Mid 2015)

$ make test-only

```

When working on an issue, you will most likely want to focus on a particular [packages](https://github.com/babel/babel/tree/main/packages). Using `TEST\_ONLY` will only run tests for that specific package.

```sh

$ TEST\_ONLY=babel-cli make test

```

`TEST\_ONLY` will also match substrings of the package name:

```sh

# Run tests for the @babel/plugin-transform-classes package.

$ TEST\_ONLY=babel-plugin-transform-classes make test

```

Use the `TEST\_GREP` variable to run a subset of tests by name:

```sh

$ TEST\_GREP=transformation make test

```

Substitute spaces for hyphens and forward slashes when targeting specific test names:

```sh

$ TEST\_GREP="arrow functions destructuring parameters" make test

```

To enable the Node.js debugger added in v6.3.0, set the `TEST\_DEBUG` environment variable:

```sh

$ TEST\_DEBUG=true make test

```

You can combine `TEST\_DEBUG` with `TEST\_GREP` or `TEST\_ONLY` to debug a subset of tests. If you plan to stay long in the debugger (which you'll likely do!), you may increase the test timeout by editing [test/testSetupFile.js](https://github.com/babel/babel/blob/main/test/testSetupFile.js).

To overwrite any test fixtures when fixing a bug or anything, add the env variable `OVERWRITE=true`

```sh

$ OVERWRITE=true TEST\_ONLY=babel-plugin-transform-classes make test-only

```

To test the code coverage, use:

```sh

$ BABEL\_ENV=cov make build

$ ./scripts/test-cov.sh

```

#### Troubleshooting Tests

In case you're not able to reproduce an error on CI locally, it may be due to

- Node Version: Travis CI runs the tests against all major node versions. If your tests use JavaScript features unsupported by lower versions of node, then use [minNodeVersion option](#writing-tests) in options.json.

- Timeout: Check the CI log and if the only errors are timeout errors and you are sure that it's not related to the changes you made, ask someone in the slack channel to trigger rebuild on the CI build and it might be resolved

In case you're locally getting errors which are not on the CI, it may be due to

- Updates in Dependencies: Make sure you run `make bootstrap` before you run `make build` or `make watch` before you run the tests.

### Writing tests

Most packages in [`/packages`](https://github.com/babel/babel/tree/main/packages) have a `test` folder, however some tests might be in other packages or in [`/packages/babel-core`](https://github.com/babel/babel/tree/main/packages/babel-core/test/fixtures).

#### `@babel/plugin-x`

All the Babel plugins (and other packages) that have a `/test/fixtures` are written in a similar way.

For example, in [`@babel/plugin-transform-exponentiation-operator/test`](https://github.com/babel/babel/tree/main/packages/babel-plugin-transform-exponentiation-operator/test):

- There is an `index.js` file. It imports our [test helper](https://github.com/babel/babel/tree/main/packages/babel-helper-plugin-test-runner). (You don't have to worry about this).

- There can be multiple folders under [`/fixtures`](https://github.com/babel/babel/tree/main/packages/babel-plugin-transform-exponentiation-operator/test/fixtures)

- There is an [`options.json`](https://github.com/babel/babel/blob/main/packages/babel-plugin-transform-exponentiation-operator/test/fixtures/exponentian-operator/options.json) file whose function is similar to a `.babelrc` file, allowing you to pass in the plugins and settings you need for your tests.

- For this test, we only need the relevant plugin, so it's just `{ "plugins": ["@babel/plugin-transform-exponentiation-operator"] }`.

- If necessary, you can have an `options.json` with different options in each subfolder.

- In each subfolder, you can organize your directory structure by categories of tests. (Example: these folders can be named after the feature you are testing or can reference the issue number they fix)

- Generally, there are two kinds of tests for plugins

  - The first is a simple test of the input and output produced by running Babel on some code. We do this by creating an [`input.js`](https://github.com/babel/babel/blob/main/packages/babel-plugin-transform-exponentiation-operator/test/fixtures/exponentian-operator/binary/input.js) file and an [`output.js`](https://github.com/babel/babel/blob/main/packages/babel-plugin-transform-exponentiation-operator/test/fixtures/exponentian-operator/binary/output.js) file. This kind of test only works in sub-subdirectories of `/fixtures`, i.e. `/fixtures/exponentian-operator/binary/input.js` and \*\*not\*\* `/fixtures/exponentian-operator/input.js`.

- If you need to expect an error, you can ignore creating the `output.js` file and pass a new `throws` key to the `options.json` that contains the error string that is created.

- The second and preferred type is a test that actually evaluates the produced code and asserts that certain properties are true or false. We do this by creating an [`exec.js`](https://github.com/babel/babel/blob/main/packages/babel-plugin-transform-exponentiation-operator/test/fixtures/exponentian-operator/comprehensive/exec.js) file.

In a fixture test, you simply write out the code you want to transform in `input.js`.

```js

// input.js

2 \*\* 2;

```

and the expected output after transforming it with your `options.json` in `output.js`.

```js

// output.js

Math.pow(2, 2);

```

In an `exec.js` test, we run or check that the code actually does what it's supposed to do rather than just check the static output.

```js

// exec.js

expect(2 \*\* 3).toBe(8);

expect(3 \* 2 \*\* 3).toBe(24);

```

If you need to check for an error that is thrown you can add to the `options.json`

```js

// options.json example

{

"plugins": [["@babel/plugin-proposal-object-rest-spread", { "useBuiltIns": "invalidOption" }]],

"throws": "@babel/plugin-proposal-object-rest-spread currently only accepts a boolean option for useBuiltIns (defaults to false)"

}

```

If the test requires a minimum Node version, you can add `minNodeVersion` (must be in semver format).

```js

// options.json example

{

"minNodeVersion": "5.0.0"

}

```

#### `@babel/parser` (babylon)

Writing tests for the babel parser is very

similar to the other packages.

Inside the `packages/babel-parser/test/fixtures` folder are categories/groupings of test fixtures (es2015, flow,

etc.). To add a test, create a folder under one of these groupings (or create a new one) with a

descriptive name, and add the following:

- Create an `input.js` file that contains the code you want the babel parser to parse.

- Add an `output.json` file with the expected parser output. For added convenience, if there is no `output.json` present, the test runner will generate one for you.

After writing tests for @babel/parser, just build it by running:

```sh

$ make build

```

Then, to run the tests, use:

```sh

$ TEST\_ONLY=babel-parser make test-only

```

#### Bootstrapping expected output

For both `@babel/plugin-x` and `@babel/parser`, you can easily generate an `output.js`/`output.json` automatically by just providing `input.js` and running the tests as you usually would.

```

// Example

- packages

- babel-parser

- test

- fixtures

- comments

- basic

- block-trailing-comment

- input.js

- output.json (will be generated if not created)

```

### Debugging code

A common approach to debugging JavaScript code is to walk through the code using the [Chrome DevTools](https://developers.google.com/web/tools/chrome-devtools/) debugger.

For illustration purposes, we are going to assume that we need to get a better understanding of [`Generator.generate()`](https://github.com/babel/babel/blob/b5246994b57f06af871be6a63dcc4c6fd41d94d6/packages/babel-generator/src/index.js#L32), which is responsible for generating code for a given AST.

To get a better understanding of what is actually going on for this particular piece of code, we are going to make use of breakpoints.

```diff

generate() {

+ debugger; // breakpoint

return super.generate(this.ast);

}

```

To include the changes, we have to make sure to build Babel:

```bash

$ make build

```

Next, we need to execute `Generator.generate()`, which can be achieved by running a test case in the `@babel/generator` package.

For example, we can run the test case that tests the generation of class declarations:

```bash

$ TEST\_DEBUG=true TEST\_GREP=ClassDeclaration make test-only

./scripts/test.sh

Debugger listening on port 9229.

To start debugging, open the following URL in Chrome:

chrome-devtools://devtools/remote/serve\_file/@60cd6e859b9f557d2312f5bf532f6aec5f284980/inspector.html?experiments=true&v8only=true&ws=127.0.0.1:9229/3cdaebd2-be88-4e7b-a94b-432950ab72d0

```

To start the debugging in Chrome DevTools, open the given URL.

The debugger starts at the first executed line of code, which is Mocha's first line by default.

Click \_Resume script execution\_ <img src="https://i.imgur.com/TmYBn9d.png" alt="Resume script execution button." width="16"> to jump to the set breakpoint.

Note that the code shown in Chrome DevTools is compiled code and therefore differs.

## Creating a new plugin (`spec-new`)

> Example: https://github.com/babel/babel/pull/11640

- Create a new PR that describes the proposed AST shape in [ESTree](https://github.com/estree/estree) (ex: [Decimal AST](https://github.com/estree/estree/pull/220)). The new AST should follow ESTree's [design philosophy](https://github.com/estree/estree#philosophy).

- After the ESTree PR is accepted, update [ast/spec.md](https://github.com/babel/babel/blob/master/packages/babel-parser/ast/spec.md). Note that there are differences between Babel AST and ESTree. In these cases, consistency with current Babel AST outweighs alignment to ESTree. Otherwise it should follow ESTree.

- [ ] Implement parser plugins based on the new AST. The parser plugin name should be the unprefixed slug of the TC39 proposal URL in \_camelcase\_, i.e. `exportDefaultFrom` from `https://github.com/tc39/proposal-export-default-from`.

- [ ] Use the `this.expectPlugin("newSyntax")` check within `@babel/parser` to ensure your new plugin code only runs when that flag is turned on (not default behavior), and a friendly error is thrown if users forget to enable a plugin.

- [ ] Add failing/passing tests according to spec behavior

- [ ] Add `@babel/syntax-new-syntax` package. You can copy `packages/babel-plugin-syntax-decimal` and replace `decimal` to `new-syntax`.

- [ ] Add `@babel/syntax-new-syntax` to `@babel/standalone`.

- [ ] Add `@babel/syntax-new-syntax` to `package.json`

- [ ] Add `@babel/syntax-new-syntax` to [`pluginsConfig.json`](https://github.com/babel/babel/blob/master/packages/babel-standalone/scripts/pluginConfig.json), run `make generate-standalone`.

- [ ] Add `@babel/syntax-new-syntax` to `src/preset-stage-x`.

- [ ] Add `"newSyntax"` to parser [typings](https://github.com/babel/babel/blob/master/packages/babel-parser/typings/babel-parser.d.ts)

- [ ] Implement generator support in `packages/babel-generator/src/generators`. The generator converts AST to source code.

- [ ] If this feature can be transpiled, start working on the Babel transform.

## Internals

- AST spec ([babel-parser/ast/spec.md](https://github.com/babel/babel/blob/main/packages/babel-parser/ast/spec.md))

- Versioning ([doc/design/versioning.md](https://github.com/babel/babel/blob/main/doc/design/versioning.md))

- Monorepo ([doc/design/monorepo.md](https://github.com/babel/babel/blob/main/doc/design/monorepo.md))

- Compiler environment support ([doc/design/compiler-environment-support.md](https://github.com/babel/babel/blob/main/doc/design/compiler-environment-support.md))

- Compiler assumptions ([doc/design/compiler-assumptions.md](https://github.com/babel/babel/blob/main/doc/design/compiler-assumptions.md))