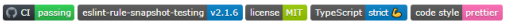
eslint-rule-snapshot-testing

2.1.6 • Public • Published a year ago



ESLint Rule Snapshot Testing

The convenience of Jest snapshot testing brought to ESLint rule authoring. First-class fixture support allows you to write input source code in its own file instead of embedded in a string in tests, avoiding indendentation and escaping issues.

Usage

1. Create a test file for your rule to be run by Jest, e.g. `my-rule-name.test.js`:

```
import { runLintFixtureTests } from 'eslint-rule-snapshot-testing';
import noFooAllowed from '../rules/no-foo-allowed';

// will generate and run tests for you
runLintFixtureTests({
    rule: noFooAllowed,
    ruleName: 'no-foo-allowed',
});
```

2. Create a fixture file for your rule tests. Name it `my-rule-name.fixture`, matching the `ruleName` passed above, and put it next to your test.

3. Write some code in the fixture file. Put a mix of valid and invalid code in it:

```
const okay = 'this will be okay';
const foo = 'this is invalid';
const okayToo = 'this will also be okay';
```

4. Run your tests with Jest. The fixture will be parsed and linted, and the results placed in a snapshot file for you with error locations visually marked in the input source. The squiggles in the snapshot will match the error underlining you would see in your editor. In this example, our rule doesn't allow 'foo' for variable names:

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`should lint correctly 1`] = `
"const okay = 'this will be okay';
const foo = 'this is invalid';
      ~~~ [1]
const okayToo = 'this will also be okay';

[1] Variable name should not be 'foo'. Pick something else."
`;
```

Advanced Usage

Splitting test cases with JSDoc

Besides source code to lint, rules also receive context like filename and rule options when running. To support this, a small JSDoc-based DSL is included to embed these inputs into fixtures. Continuing our example above, let's say our rule adds an option to customize the forbidden variable names, and we've also adjusted our rule to ignore test files. We can split our fixture into test cases to cover this functionality, still contained in a single file.

`my-rule-name.fixture`

```
/**
 * @test fails with foo by default
 */
const okay = 'something';
const foo = 'fails';

/**
 * @test allows overriding forbidden variable names
 * @ruleOptions [{ "forbidden": ["bar", "baz"] }]
 */
const bar = 'fails';
const baz = 'also fails';
const foo = 'this is now okay';

/**
 * @test allows forbidden variable names in tests
 * @filename something.spec.js
 */
const foo = 'this is now okay';

/**
 * @test allows overridden variable names in tests
 * @filename something.spec.js
```

Install

```
> npm i eslint-rule-snapshot-testing
```

Repository
github.com/jwbay/eslint-snapshot-testing

Homepage
github.com/jwbay/eslint-snapshot-testi...

Weekly Downloads
0

Version 2.1.6
License MIT

Unpacked Size 61.1 kB
Total Files 21

Issues 0
Pull Requests 3

Last publish
a year ago

Collaborators

Try on RunKit

Report malware

```
 * @ruleOptions [{ forbidden: [ 'bar', 'baz' ] }]
 */
const bar = 'okay';
const baz = 'also okay';
```

The fixture will be parsed and split into discrete tests:

```
 PASS  src/tests/my-rule-name.test.ts
  √ fails with foo by default (24 ms)
  √ allows overriding forbidden variable names (2 ms)
  √ allows forbidden variable names in tests (2 ms)
  √ allows overridden variable names in tests (1 ms)
```

And corresponding snapshots are written:

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`allows forbidden variable names in tests 1`] = `
"
const foo = 'this is now okay';

"
`;

exports[`allows overridden variable names in tests 1`] = `
"
const bar = 'okay';
const baz = 'also okay';
"
`;

exports[`allows overriding forbidden variable names 1`] = `
"
const bar = 'fails';
      ~~~ [1]
const baz = 'also fails';
      ~~~ [2]
const foo = 'this is now okay';


[1] variable name 'bar' should not include 'bar'.
[2] variable name 'baz' should not include 'baz'."
`;

exports[`fails with foo by default 1`] = `
"
const okay = 'something';
const foo = 'fails';
      ~~~ [1]


[1] variable name 'foo' should not include 'foo'."
`;
```

Testing fixes

Use the `acceptFix` JSDoc tag for a test to run your rule's fixer against the source code. The snapshot will contain both the before and after code.

Example: `my-autofix-rule.fixture`

```
/**
 * @test replaces foo with bar
 * @acceptFix
 */
const somethingFoo = 'something';
```

If your rule replaces text "foo" with "bar" in variables (note: renaming variables is generally unsafe for a linter and should not be done in a real rule), the following snapshot will be generated:

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`replaces foo with bar`] = `
"Original code:
========================

const somethingFoo = 'something';
      ~~~~~~~~~~~~ [1]


[1] variable name 'somethingFoo' should not include 'foo'.

Code after applying fixes:
==========================

const somethingBar = 'something'

"
`;
```

If the rule still reports errors after your fixer runs, those errors are serialized just like the initial errors.

> Note: at time of writing, ESLint runs fixes in a finite loop to allow fixes across rules to stabilize. An unstable or incomplete rule fixer may still report errors.

Syntax highlighting in fixtures

Fixture names ending in any extension are supported. Examples of valid fixture names:

- my-rule-name.fixture
- my-rule-name.fixture.js
- my-rule-name.fixture.ts
- my-rule-name.fixture.tsx
- my-rule-name.fixture.whatever

This means it's possible to get syntax highlighting, intellisense, and even type checking in your fixtures, if desired. The tradeoff is that this may be too much validation for "test" code. You may find a middle ground by ignoring fixture files from linting, formatting, or typechecking via configuration in those tools.

Fixture locations

By default, fixtures are looked for in the same directory as the calling test, or in a `__fixtures__` directory under the test directory. The test location is inferred from a generated stacktrace, so location inference may break in certain cases. You can always set the fixture location for tests manually, even if just to override it. The fixture location must be an absolute path.

```
runLintFixtureTests({
    rule: myRule,
    ruleName: 'my-rule-name',
    fixtureDirectory: path.join(__dirname, 'custom-fixture-directory'),
});
```

ESLint options

ESLint-level configuration can be set when running fixture tests. You can use this to adjust parser options for your rule, for example.

```
runLintFixtureTests({
    rule: myRule,
    ruleName: 'my-rule-name',
    eslintConfig: {
        parserOptions: {
            ecmaVersion: 2020,
            sourceType: 'module',
        },
    },
});
```

Full control over tests with the raw serializer

One limitation of the fixture file approach, even with the DSL, is external inputs that can be consulted during rule execution -- custom configuration files, current working directory, etc. To account for this, the underlying snapshot serializer is exported separately. You can use this along with calling the ESLint APIs directly in your test to get full control over execution, allowing for per-test mocking or other setup. As an example, we can imagine a rule which consults `process.cwd()` to determine whether something is a lint error or not:

```
import { Linter } from 'eslint';
import { serializeLintResult } from 'eslint-rule-snapshot-testing';
import { myRule } from '../rules/my-rule';

const processCwd = jest.spyOn(process, 'cwd');
const sourceCode = `\nconst foo = 'something';`;

describe('serializeLintResult supports per-test setup and mocking for lint rules'
    test('should not have errors', () => {
        processCwd.mockReturnValue('/foo');
        expect(lint(sourceCode)).toMatchSnapshot();
    });

    test('should have errors', () => {
        processCwd.mockReturnValue('/bar');
        expect(lint(sourceCode)).toMatchSnapshot();
    });
});

function lint(source) {
    const linter = new Linter({});
    linter.defineRule('my-rule-name', testRule);
    const lintMessages = linter.verify(source, { rules: { ['my-rule-name']: 'erro
    return serializeLintResult({
        lintedSource: source,
        lintMessages,
    });
}
```

Limitations and tradeoffs

- Double quotes in source code are subject to noise in snapshots due to escaping
- Testing lint rule suggestions is not yet supported

Keywords

jest eslint rules snapshot test fixture