postcss-bem-linter

build unknown

A PostCSS plugin to lint *BEM-style* CSS.

BEM-style describes CSS that follows a more-or-less strict set of conventions determining what selectors can be used. Typically, these conventions require that classes begin with the name of the component (or "block") that contains them, and that all characters after the component name follow a specified pattern. Original BEM methodology refers to "blocks", "elements", and "modifiers"; SUIT refers to "components", "descendants", and "modifiers". You might have your own terms for similar concepts.

With this plugin, you can check the validity of selectors against a set of BEM-style conventions. You can use preset patterns (SUIT and BEM, currently) or insert your own. The plugin will register warnings if it finds CSS that does not follow the specified conventions.

Installation

```
npm install postcss postcss-bem-linter --save-dev
```

Version 1.0.0+ is compatible with PostCSS 5+. (Earlier versions are compatible with PostCSS 4.)

This plugin registers warnings via PostCSS. Therefore, you'll want to use it with a PostCSS runner that prints warnings (e.g. `gulp-postcss`) or another PostCSS plugin that prints warnings (e.g. `postcss-reporter`).

Throughout this document, terms like "selector", "selector sequence", and "simple selector" are used according to the definitions in the Selectors Level 3 spec.

stylelint plugin

postcss-bem-linter can also be used as a stylelint plugin: stylelint-selector-bem-pattern.

By using the stylelint plugin, all of your linting can happen in one step, seamlessly: postcss-bem-linter warnings will output alongside other stylelint warnings. Also, you can take advantage of all the other features that stylelint offers, such as a CLI and Node.js API, different formatters for output, etc.

Conformance tests

Default mode:

- Only allow selector sequences that match the defined convention.
- Only allow custom-property names that *begin* with the defined `ComponentName`.

Weak mode:

- While *initial* selector sequences (before combinators) must match the defined convention, sequences *after combinators* are not held to any standard.

Prior to 0.5.0, this plugin checked two other details: that `:root` rules only contain custom-properties; and that the `:root` selector is not grouped or combined with other selectors. These checks can now be performed by stylelint. So from 0.5.0 onwards, this plugin leaves that business to stylelint to focus on its more unique task.

Use

```
bemLinter([pattern[, options]])
```

Defining your pattern

Patterns consist of regular expressions, and functions that return regular expressions, or strings, which describe valid selector sequences.

Keep in mind: - **Patterns describe sequences, not just simple selectors.** So if, for example, you would like to be able to chain state classes to your component classes, as in `.Component.is-open`, your pattern needs to allow for this chaining. - **Pseudo-classes and pseudo-elements will be ignored if they occur at the end of the sequence.** Instead of `.Component:first-child.is-open`, you should use `.Component.is-open:first-child`. The former will trigger a warning unless you've written a silly complicated pattern.

Preset Patterns The following preset patterns are available:

- `'suit'` (default), as defined here. Options:
  - `namespace`: a namespace to prefix valid classes, as described in the SUIT docs
- `'bem'`, as defined here.
  - `namespace`: a namespace to prefix valid classes, to be separated from the block name with a hyphen, e.g. with namespace `foo`, `.foo-dropdown__menu`.

You can use a preset pattern and its options in two ways: - Pass the preset's name as the first argument, and, if needed, an `options` object as the second, e.g. `bemLinter('suit', { namespace: 'twt'})`. - Pass an object as the first and only argument, with the preset's name as the `preset` property and, if needed, `presetOptions`, e.g. `bemLinter({ preset: 'suit', presetOptions: { namespace: 'twt'})`.

`'suit'` is the default pattern; so if you do not pass any `pattern` argument, SUIT conventions will be enforced.

Custom Patterns You can define a custom pattern by passing as your first and only argument *an object with the following properties*:

`componentName` default: `/^[-_a-zA-Z0-9]+$/`

Describes valid component names in one of the following forms:

- A regular expression.
- A string that provides a valid pattern for the `RegExp()` constructor.

`componentSelectors` Describes all valid selector sequences for the stylesheet in one of the following forms:

- A *single function* that accepts a component name and returns a regular expression, e.g.

```
componentSelectors(componentName) {
  return new RegExp('^\\.ns-' + componentName + '(?:-[a-zA-Z]+)?$');
}
```

- A *single string* that provides a valid pattern for the `RegExp()` constructor *when {`componentName`} is interpolated with the defined component's name*, e.g.

```
componentSelectors: '^\\.ns-{componentName}(?:-[a-zA-Z]+)?$'
```

- An *object consisting of two properties*, `initial` and `combined`. Both properties accept the same two forms described above: a function accepting a component name and returning a regular expression; or a string, interpolating the component name with `{componentName}`, that will provide a valid pattern for the `RegExp()` constructor.

  `initial` describes valid initial selector sequences — those occurring at the beginning of a selector, before any combinators.

  `combined` describes valid selector sequences allowed *after* combinators. Two important notes about `combined`:

  - If you do not specify a `combined` pattern, it is assumed that combined sequences must match the same pattern as initial sequences.
  - In weak mode, *any* combined sequences are accepted, even if you have a `combined` pattern.

`utilitySelectors` Describes valid utility selector sequences. This will be used if the stylesheet defines a group of utilities, as explained below. Can take one of the following forms:

- A regular expression.
- A string that provides a valid pattern for the `RegExp()` constructor.

`ignoreSelectors` Describes selector sequences to ignore. You can use this to systematically ignore selectors matching certain patterns, instead of having to add a `/* postcss-bem-linter: ignore */` comment above each one (see below). Can take one of the following forms:

- A regular expression.
- An array of regular expressions.
- A string that provides a valid pattern for the `RegExp()` constructor.
- An array of such string patterns.

`ignoreCustomProperties` Describes custom properties to ignore. Works the same as `ignoreSelectors`, above, so please read about that.

Overriding Presets

You can also choose a preset to start with and override specific parts of it, specific patterns.

For example, if you want to use SUIT's preset generally but write your own `utilitySelectors` pattern, you can do that with a config object like this:

```
{
  preset: 'suit',
  utilitySelectors: /^\.fancyUtilities-[a-z]+$/
}
```

Examples

Given all of the above, you might call the plugin in any of the following ways:

```
// use 'suit' conventions
bemLinter();
bemLinter('suit');
bemLinter('suit', { namespace: 'twt' });
bemLinter({ preset: 'suit', presetOptions: { namespace: 'twt' }});

// use 'bem' conventions
bemLinter('bem');
bemLinter('bem', { namespace: 'ydx' });
bemLinter({ preset: 'bem', presetOptions: { namespace: 'ydx' }});

// define a pattern for component names
bemLinter({
  componentName: /^[A-Z]+$/
});
bemLinter({
  componentName: '^[A-Z]+$'
});

// define a single pattern for all selector sequences, initial or
    combined
bemLinter({
  componentSelectors(componentName) {
    return new RegExp('^\\.' + componentName + '(?:-[a-z]+)?$');
  }
});
bemLinter({
  componentSelectors: '^\\.{componentName}(?:-[a-z]+)?$'
});

// define separate `componentName`, `initial`, `combined`, and `
    utilities` patterns
bemLinter({
  componentName: /^[A-Z]+$/,
  componentSelectors: {
    initial(componentName) {
      return new RegExp('^\\.' + componentName + '(?:-[a-z]+)?$');
    },
    combined(componentName) {
      return new RegExp('^\\.combined-' + componentName + '-[a-z]+$');
    }
  },
  utilitySelectors: /^\.util-[a-z]+$/
});
bemLinter({
  componentName: '^[A-Z]+$',
  componentSelectors: {
    initial: '^\\.{componentName}(?:-[a-z]+)?$',
    combined: '^\\.combined-{componentName}-[a-z]+$'
  },
  utilitySelectors: '^\.util-[a-z]+$'
});

// start with the `bem` preset but include a special `componentName`
    pattern
// and `ignoreSelectors` pattern to ignore Modernizr-injected `no-*`
    classes
bemLinter({
  preset: 'bem',
  componentName: /^cmpnt_[a-zA-Z]+$/,
  ignoreSelectors: /^\.no-.+$/
});
bemLinter({
  preset: 'bem',
  componentName: '^cmpnt_[a-zA-Z]+$',
  ignoreSelectors: '^\.no-.+$'
});

// ... using an array for `ignoreSelectors`
bemLinter({
  preset: 'bem',
  componentName: /^cmpnt_[a-zA-Z]+$/,
  ignoreSelectors: [
    /^\.no-.+$/,
    /^\.isok-.+$/
  ]
});
bemLinter({
  preset: 'bem',
  componentName: '^cmpnt_[a-zA-Z]+$',
  ignoreSelectors: [
    '^\.no-.+$',
    '^\.isok-.+$'
  ]
});
```

Defining a component and utilities

The plugin will only lint the CSS if it knows the context of the CSS: is it a utility or a component. To define the context, use the configuration options to define it based on the filename (`css/components/*.css`) or use a special comment to define context for the CSS after it. When defining a component, the component name is needed.

Define components and utilities implicitly based on their filename When defining a component base on the filename, the name of the file (minus the extension) will be used implicitly as the component name for linting. The configuration option for implicit components take:

- Enable it for all files: `implicitComponents: true`
- Enable it for files that match a glob pattern: `implicitComponents: 'components/**/*.css'`
- Enable it for files that match one of multiple glob patterns: `implicitComponents: ['components/**/*.css', 'others/**/*.css']`

The CSS will implicitly be linted as utilities in files marked as such by their filename. The configuration option for implicit utilities take:

- Enable it for files that match a glob pattern: `implicitUtilities: 'utils/*.css'`
- Enable it for files that match one of multiple glob patterns: `implicitUtilities: ['util/*.css', 'bar/**/*.css']`

Define components/utilities with a comment These comment definitions can be provided in two syntaxes: concise and verbose.

- Concise definition syntax: `/** @define ComponentName */` or `/** @define utilities */`
- Verbose definition syntax: `/* postcss-bem-linter: define ComponentName */` or `/* postcss-bem-linter: define utilities */`.

Weak mode is turned on by adding `; weak` to a definition, e.g. `/** @define ComponentName ; weak */` or `/* postcss-bem-linter: define ComponentName; weak */`.

Concise syntax:

```css
/** @define MyComponent */

:root {
  --MyComponent-property: value;
}

.MyComponent {}

.MyComponent-other {}
```

Verbose syntax:

```css
/** postcss-bem-linter: define FancyComponent */

:root {
  --FancyComponent-property: value;
}

.FancyComponent {}
```

```

.FancyComponent-other {}
```

Weak mode:

```
/** @define MyComponent; weak */

:root {
  --MyComponent-property: value;
}

.MyComponent {}

.MyComponent .other {}
```

Implicit:

```
bemLinter({
  preset: 'bem',
  implicitComponents: 'components/**/*.css',
  implicitUtilities: 'utils/*.css'
});
```

Utilities:

```
/** @define utilities */

.u-sizeFill {}

.u-sm-horse {}
```

If a component is defined and the component name does not match your `componentName` pattern, the plugin will throw an error.

Multiple definitions

It's recommended that you keep each defined group of rules in a distinct file, with the definition at the top of the file. If, however, you have a good reason for *multiple definitions within a single file*, you can do that.

Successive definitions override each other. So the following works:

```
/** @define Foo */
.Foo {}

/** @define Bar */
.Bar {}

```

```css
/** @define utilities */
.u-something {}
```

You can also deliberately *end the enforcement of a definition* with the following special comments: `/** @end */` or `/* postcss-bem-linter: end */`.

```css
/** @define Foo */
.Foo {}
/** @end */

.something-something-something {}
```

One use-case for this functionality is when linting files *after* concatenation performed by a CSS processor like Less or Sass, whose syntax is not always compatible with PostCSS. See issue #57.

Ignoring specific selectors

If you need to ignore a specific selector but do not want to ignore the entire stylesheet or end the enforcement of a definition, there are two ways to accomplish this:

As describe above, you can include an `ignoreSelectors` regular expression (or array of regular expressions) in your configuration. This is the best approach if you want to systematically ignore all selectors matching a pattern (e.g. all Modernizr classes).

If you just want to ignore a single, isolated selector, though, you can do so by *preceding the selector* with this comment: `/* postcss-bem-linter: ignore */`.

```css
/** @define MyComponent */

.MyComponent {
  display: flex;
}

/* postcss-bem-linter: ignore */
.no-flexbox .Component {
  display: block;
}
```

*The comment will cause the linter to ignore **only** the very next selector.*

Testing CSS files

Pass your individual CSS files through the plugin. It will register warnings for conformance failures, which you can print to the console using `postcss-reporter` or relying on a PostCSS runner (such as `gulp-postcss`).

```
const postcss = require('postcss');
const bemLinter = require('postcss-bem-linter');
const reporter = require('postcss-reporter');

files.forEach(file => {
  const css = fs.readFileSync(file, 'utf-8');
  postcss()
    .use(bemLinter())
    .use(reporter())
    .process(css)
    .then(result => { .. });
});
```

Contributing

Please note that this project is released with a Contributor Code of Conduct. By participating in this project you agree to abide by its terms.

Development

Install dependencies. Requires Yarn 1.x (Classic)

```
yarn
```

Run the tests.

```
yarn test
```

Watch and automatically re-run the unit tests.

```
yarn start
```