License

clean-css is released under the MIT License.

clean-css logo

clean-css is a fast and efficient CSS optimizer for Node.js platform and any modern browser.

According to tests it is one of the best available.

Table of Contents

Node.js version support
Install
Use
What's new in version 5.3
What's new in version 5.0
What's new in version 4.2
What's new in version 4.1
Important: 4.0 breaking changes
Constructor options
Compatibility modes
Fetch option
Formatting options
Inlining options

- Optimization levels
  - Level 0 optimizations
  - Level 1 optimizations
  - Level 2 optimizations
- Plugins
- Minify method
- Promise interface
- CLI utility
- FAQ
  - How to optimize multiple files?
  - How to process multiple files without concatenating them into one output file?
  - How to process remote `@imports` correctly?
  - How to apply arbitrary transformations to CSS properties?
  - How to specify a custom rounding precision?
  - How to keep a CSS fragment intact?
  - How to preserve a comment block?
  - How to rebase relative image URLs?
  - How to work with source maps?
  - How to apply level 1 & 2 optimizations at the same time?
  - What level 2 optimizations do?
  - What errors and warnings are?
  - How to use clean-css with build tools?
  - How to use clean-css from web browser?
- Contributing
  - How to get started?
- Acknowledgments
- License

- @wagenet (Peter Wagenet) for suggesting improvements to `@import` inlining behavior;
- @venemo (Timur Kristóf) for an outstanding contribution of advanced property optimizer for 2.2 release;
- @vvo (Vincent Voyer) for a patch with better empty element regex and for inspiring us to do many performance improvements in 0.4 release;
- @xhmikosr for suggesting new features, like option to remove special comments and strip out URLs quotation, and pointing out numerous improvements like JSHint, media queries, etc.

Acknowledgments

Sorted alphabetically by GitHub handle:

- @abarre (Anthony Barre) for improvements to `@import` processing;
- @alexlamsl (Alex Lam S.L.) for testing early clean-css 4 versions, reporting bugs, and suggesting numerous improvements.
- @altschuler (Simon Altschuler) for fixing `@import` processing inside comments;
- @ben-eb (Ben Briggs) for sharing ideas about CSS optimizations;
- @davisjam (Jamie Davis) for disclosing ReDOS vulnerabilities;
- @facelessuser (Isaac) for pointing out a flaw in clean-css' stateless mode;
- @grandrath (Martin Grandrath) for improving `minify` method source traversal in ES6;
- @jmalonzo (Jan Michael Alonzo) for a patch removing node.js' old `sys` package;
- @lukeapage (Luke Page) for suggestions and testing the source maps feature; Plus everyone else involved in #125 for pushing it forward;
- @madwizard-thomas for sharing ideas about `@import` inlining and URL rebasing.
- @ngyikp (Ng Yik Phang) for testing early clean-css 4 versions, reporting bugs, and suggesting numerous improvements.

Node.js version support

clean-css requires Node.js 10.0+ (tested on Linux, OS X, and Windows)

Install

```
npm install --save-dev clean-css
```



```
npm run check # to lint JS sources with
        [JSHint](https://github.com/
        jshint/jshint/)
npm test # to run all tests
```

Contributing

See CONTRIBUTING.md.

How to get started?

First clone the sources:

```
git clone git@github.com:clean-css/
        clean-css.git
```

then install dependencies:

```
cd clean-css
npm install
```

then use any of the following commands to verify your copy:

```
npm run bench
        # for clean-css benchmarks (see
        [test/bench.js](https://
        github.com/clean-css/clean-css/
        blob/master/test/bench.js) for
        details)
npm run browserify # to create the
        browser-ready clean-css version
```


Use

```
var CleanCSS = require('clean-css');
var input = 'a{font-weight:bold;}';
var options = { /* options */ };
var output = new
        CleanCSS(options).minify(input);
```

What's new in version 5.3

clean-css 5.3 introduces one new feature:

- variables can be optimized using level 1's `variableValueOptimizers` option, which accepts a list of value optimizers or a list of their names, e.g. `variableValueOptimizers: ['color', 'fraction']`.

What's new in version 5.0

clean-css 5.0 introduced some breaking changes:

- Node.js 6.x and 8.x are officially no longer supported;
- `transform` callback in level-1 optimizations is removed in favor of new plugins interface;

- changes default Internet Explorer compatibility from 10+ to >11, to revert the old default use `{ compatibility: 'ie10' }` flag;
- changes default `rebase` option from `true` to `false` so URLs are not rebased by default. Please note that if you set `rebaseTo` option it still counts as setting `rebase: true` to preserve some of the backward compatibility.

And on the new features side of things:

- format options now accepts numerical values for all breaks, which will allow you to have more control over output formatting, e.g. `format: {breaks: {afterComment: 2}}` means clean-css will add two line breaks after each comment
- a new `batch` option (defaults to `false`) is added, when set to `true` it will process all inputs, given either as an array or a hash, without concatenating them.

What's new in version 4.2

clean-css 4.2 introduces the following changes / features:

- Adds `process` method for compatibility with optimize-css-assets-webpack-plugin;
- new `transition` property optimizer;
- preserves any CSS content between `/* clean-css ignore:start */` and `/* clean-css ignore:end */` comments;
- allows filtering based on selector in `transform` callback, see example;

- http://refresh-sf.com/
- http://adamburgess.github.io/clean-css-online/

```
            const buferFile = new
        CleanCSS(options).minify(file.contents)
            return file.contents =
        Buffer.from(buferFile.styles)
        })
        .pipe(dest('build'))
}
exports.css = series(css)
```

How to use clean-css with build tools?

There is a number of 3rd party plugins to popular build tools:

- Broccoli: broccoli-clean-css
- Brunch: clean-css-brunch
- Grunt: grunt-contrib-cssmin
- Gulp: gulp-clean-css
- Gulp: using vinyl-map as a wrapper - courtesy of @sogko
- component-builder2: builder-clean-css
- Metalsmith: metalsmith-clean-css
- Lasso: lasso-clean-css
- Start: start-clean-css

How to use clean-css from web browser?

- https://clean-css.github.io/ (official web interface)

- adds configurable line breaks via `format: { breakWith: 'lf' }` option.

What's new in version 4.1

clean-css 4.1 introduces the following changes / features:

- `inline: false` as an alias to `inline: ['none']`;
- `multiplePseudoMerging` compatibility flag controlling merging of rules with multiple pseudo classes / elements;
- `removeEmpty` flag in level 1 optimizations controlling removal of rules and nested blocks;
- `removeEmpty` flag in level 2 optimizations controlling removal of rules and nested blocks;
- `compatibility: { selectors: { mergeLimit: <number> } }` flag in compatibility settings controlling maximum number of selectors in a single rule;
- `minify` method improved signature accepting a list of hashes for a predictable traversal;
- `selectorsSortingMethod` level 1 optimization allows `false` or `'none'` for disabling selector sorting;
- `fetch` option controlling a function for handling remote requests;
- new `font` shorthand and `font-*` longhand optimizers;
- removal of `optimizeFont` flag in level 1 optimizations due to new `font` shorthand optimizer;
- `skipProperties` flag in level 2 optimizations controlling which properties won't be optimized;

- new `animation` shorthand and `animation-*` longhand optimizers;
- `removeUnusedAtRules` level 2 optimization controlling removal of unused `@counter-style`, `@font-face`, `@keyframes`, and `@namespace` at rules;
- the web interface gets an improved settings panel with “reset to defaults”, instant option changes, and settings being persisted across sessions.

Important: 4.0 breaking changes

clean-css 4.0 introduces some breaking changes:

- API and CLI interfaces are split, so API stays in this repository while CLI moves to clean-css-cli;
- `root`, `relativeTo`, and `target` options are replaced by a single `rebaseTo` option - this means that rebasing URLs and import inlining is much simpler but may not be (YMMV) as powerful as in 3.x;
- `debug` option is gone as stats are always provided in output object under `stats` property;
- `roundingPrecision` is disabled by default;
- `roundingPrecision` applies to **all** units now, not only `px` as in 3.x;
- `processImport` and `processImportFrom` are merged into `inline` option which defaults to `local`. Remote `@import` rules are **NOT** inlined by default anymore;

Clean-css for Gulp

An example of how you can include clean-css in gulp

```
const { src, dest, series } =
        require('gulp');
const CleanCSS = require('clean-css');
const concat = require('gulp-concat');

function css() {
    const options = {
        compatibility:
        '*', // (default) - Internet
        Explorer 10+ compatibility mode
        inline: ['all'], // enables all
        inlining, same as ['local',
        'remote']
        level:
        2 // Optimization levels. The
        level option can be either 0, 1
        (default), or 2, e.g.
        // Please note that level 1
        optimization options are
        generally safe while level 2
        optimizations should be safe for
        most users.
    };

    return src('app/**/*.css')
        .pipe(concat('style.min.css'))
        .on('data', function(file) {
```

```
  @import "idontexist.css";
  a {
    color: blue;
  }
  div {
    margin: 5px
  }

`);

console.log(output);

// Log:
{
  styles: 'a{color:#00f}
       div{margin:5px}',
  stats: {
    efficiency: 0.7627118644067796,
    minifiedSize: 28,
    originalSize: 118,
    timeSpent: 2
  },
  errors: [
    'Ignoring local @import of
        "idontexist.css" as resource is
        missing.'
  ],
  inlinedStylesheets: [],
  warnings: []
}
```


- splits `inliner: { request: ..., timeout: ... }` option into `inlineRequest` and `inlineTimeout` options;
- remote resources without a protocol, e.g. `//fonts.googleapis.com/css?family=Domine:700`, are not inlined anymore;
- changes default Internet Explorer compatibility from 9+ to 10+, to revert the old default use `{ compatibility: 'ie9' }` flag;
- renames `keepSpecialComments` to `specialComments`;
- moves `roundingPrecision` and `specialComments` to level 1 optimizations options, see examples;
- moves `mediaMerging`, `restructuring`, `semanticMerging`, and `shorthandCompacting` to level 2 optimizations options, see examples below;
- renames `shorthandCompacting` option to `mergeIntoShorthands`;
- level 1 optimizations are the new default, up to 3.x it was level 2;
- `keepBreaks` option is replaced with `{ format: 'keep-breaks' }` to ease transition;
- `sourceMap` option has to be a boolean from now on - to specify an input source map pass it a 2nd argument to `minify` method or via a hash instead;
- `aggressiveMerging` option is removed as aggressive merging is replaced by smarter override merging.

Constructor options

clean-css constructor accepts a hash as a parameter with the following options available:

- `compatibility` - controls compatibility mode used; defaults to `ie10+`; see compatibility modes for examples;
- `fetch` - controls a function for handling remote requests; see fetch option for examples (since 4.1.0);
- `format` - controls output CSS formatting; defaults to `false`; see formatting options for examples;
- `inline` - controls `@import` inlining rules; defaults to `'local'`; see inlining options for examples;
- `inlineRequest` - controls extra options for inlining remote `@import` rules, can be any of HTTP(S) request options;
- `inlineTimeout` - controls number of milliseconds after which inlining a remote `@import` fails; defaults to 5000;
- `level` - controls optimization level used; defaults to `1`; see optimization levels for examples;
- `rebase` - controls URL rebasing; defaults to `false`;
- `rebaseTo` - controls a directory to which all URLs are rebased, most likely the directory under which the output file will live; defaults to the current directory;
- `returnPromise` - controls whether `minify` method returns a Promise object or not; defaults to `false`; see promise interface for examples;
- `sourceMap` - controls whether an output source map is built; defaults to `false`;


```
  div {
    margin: 5px
  }

`);

console.log(output);

// Log:
{
  styles: 'div{margin:5px}',
  stats: {
    efficiency: 0.8695652173913043,
    minifiedSize: 15,
    originalSize: 115,
    timeSpent: 1
  },
  errors: [],
  inlinedStylesheets: [],
  warnings: [
    "Invalid property name '-
        notarealproperty-' at 4:8.
        Ignoring.",
    "Empty property 'color' at 5:8.
        Ignoring."
  ]
}
```

Example: Minify invalid CSS, resulting in one error:

```
const CleanCSS = require("clean-css");

const output = new CleanCSS().minify(`
```

e.g. `.one{padding:0}.two{margin:0}.one{margin-bottom:3px}` into `.two{margin:0}.one{padding:0;margin-bottom:3px}`;

- `removeDuplicateFontAtRules` - removes duplicated `@font-face` rules;
- `removeDuplicateMediaQueries` - removes duplicated `@media` nested blocks;
- `mergeMediaQueries` - merges non-adjacent `@media` at-rules by the same rules as `mergeNonAdjacentBy*` above;

What errors and warnings are?

If clean-css encounters invalid CSS, it will try to remove the invalid part and continue optimizing the rest of the code. It will make you aware of the problem by generating an error or warning. Although clean-css can work with invalid CSS, it is always recommended that you fix warnings and errors in your CSS.

Example: Minify invalid CSS, resulting in two warnings:

```
const CleanCSS = require("clean-css");

const output = new CleanCSS().minify(`

  a {
    -notarealproperty-: 5px;
    color:
  }
```


- `sourceMapInlineSources` - controls embedding sources inside a source map's `sourcesContent` field; defaults to false.

Compatibility modes

There is a certain number of compatibility mode shortcuts, namely:

- `new CleanCSS({ compatibility: '*' })` (default) - Internet Explorer 10+ compatibility mode
- `new CleanCSS({ compatibility: 'ie9' })` - Internet Explorer 9+ compatibility mode
- `new CleanCSS({ compatibility: 'ie8' })` - Internet Explorer 8+ compatibility mode
- `new CleanCSS({ compatibility: 'ie7' })` - Internet Explorer 7+ compatibility mode

Each of these modes is an alias to a fine grained configuration, with the following options available:

```
new CleanCSS({
  compatibility: {
    colors: {
      hexAlpha: false, // controls 4-
        and 8-character hex color
        support
      opacity: true // controls
        `rgba()` / `hsla()` color
        support
    },
    properties: {
```

```
backgroundClipMerging: true, //
  controls background-clip merging
  into shorthand
backgroundOriginMerging: true, //
  controls background-origin
  merging into shorthand
backgroundSizeMerging: true, //
  controls background-size merging
  into shorthand
colors: true, // controls color
  optimizations
ieBangHack: false, // controls
  keeping IE bang hack
ieFilters: false, // controls
  keeping IE `filter` / `-ms-
  filter`
iePrefixHack: false, // controls
  keeping IE prefix hack
ieSuffixHack: false, // controls
  keeping IE suffix hack
merging: true, // controls
  property merging based on
  understandability
shorterLengthUnits: false, //
  controls shortening pixel units
  into `pc`, `pt`, or `in` units
spaceAfterClosingBrace: true, //
  controls keeping space after
  closing brace - `url() no-
  repeat` into `url()no-repeat`
urlQuotes: true, // controls
  keeping quoting inside `url()`
zeroUnits: true // controls
  removal of units `0` value
```


What level 2 optimizations do?

All level 2 optimizations are dispatched here, and this is what they do:

- `recursivelyOptimizeBlocks` - does all the following operations on a nested block, like `@media` or `@keyframe`;
- `recursivelyOptimizeProperties` - optimizes properties in rulesets and flat at-rules, like @font-face, by splitting them into components (e.g. `margin` into `margin-(bottom|left|right|top)`), optimizing, and restoring them back. You may want to use `mergeIntoShorthands` option to control whether you want to turn multiple components into shorthands;
- `removeDuplicates` - gets rid of duplicate rulesets with exactly the same set of properties, e.g. when including a Sass / Less partial twice for no good reason;
- `mergeAdjacent` - merges adjacent rulesets with the same selector or rules;
- `reduceNonAdjacent` - identifies which properties are overridden in same-selector non-adjacent rulesets, and removes them;
- `mergeNonAdjacentBySelector` - identifies same-selector non-adjacent rulesets which can be moved (!) to be merged, requires all intermediate rulesets to not redefine the moved properties, or if redefined to have the same value;
- `mergeNonAdjacentByBody` - same as the one above but for same-selector non-adjacent rulesets;
- `restructure` - tries to reorganize different-selector different-rules rulesets so they take less space,

```
    sourceMap: '...source-map...'
  },
  'path/to/source/2': {
    styles: '...styles...',
    sourceMap: '...source-map...'
  }
}, function (error, output) {
  // access output.sourceMap as above
});
```

How to apply level 1 & 2 optimizations at the same time?

Using the hash configuration specifying both optimization levels, e.g.

```
new CleanCSS({
  level: {
    1: {
      all: true,
      normalizeUrls: false
    },
    2: {
      restructureRules: true
    }
  }
})
```

will apply level 1 optimizations, except url normalization, and default level 2 optimizations with rule restructuring.


```
},
selectors: {
  adjacentSpace: false, // controls
    extra space before `nav` element
  ie7Hack:
    true, // controls removal of IE7
    selector hacks, e.g. `*+html...`
  mergeablePseudoClasses:
    [':active', ...], // controls a
    whitelist of mergeable pseudo
    classes
  mergeablePseudoElements:
    ['::after', ...], // controls a
    whitelist of mergeable pseudo
    elements
  mergeLimit: 8191, // controls
    maximum number of selectors in a
    single rule (since 4.1.0)
  multiplePseudoMerging: true //
    controls merging of rules with
    multiple pseudo classes /
    elements (since 4.1.0)
},
units: {
  ch: true, // controls treating
    `ch` as a supported unit
  in: true, // controls treating
    `in` as a supported unit
  pc: true, // controls treating
    `pc` as a supported unit
  pt: true, // controls treating
    `pt` as a supported unit
  rem: true, // controls treating
    `rem` as a supported unit
```

```
      vh: true, // controls treating
        `vh` as a supported unit
      vm: true, // controls treating
        `vm` as a supported unit
      vmax: true, // controls treating
        `vmax` as a supported unit
      vmin: true // controls treating
        `vmin` as a supported unit
    }
  }
})
```

You can also use a string when setting a compatibility mode, e.g.

```
new CleanCSS({
  compatibility: 'ie9,-
        properties.merging' // sets
        compatibility to IE9 mode with
        disabled property merging
})
```

Fetch option

The `fetch` option accepts a function which handles remote resource fetching, e.g.

```
var request = require('request');
var source = '@import url(http://
        example.com/path/to/
        stylesheet.css);';
```



```
new CleanCSS({ sourceMap: true,
        rebaseTo:
        pathToOutputDirectory })
  .minify(source, function (error,
        output) {
    // access output.sourceMap for
        SourceMapGenerator object
    // see https://github.com/mozilla/
        source-map/#sourcemapgenerator
        for more details
});
```

You can also pass an input source map directly as a 2nd argument to `minify` method:

```
new CleanCSS({ sourceMap: true,
        rebaseTo:
        pathToOutputDirectory })
  .minify(source, inputSourceMap,
        function (error, output) {
    // access output.sourceMap to access
        SourceMapGenerator object
    // see https://github.com/mozilla/
        source-map/#sourcemapgenerator
        for more details
});
```

or even multiple input source maps at once:

```
new CleanCSS({ sourceMap: true,
        rebaseTo:
        pathToOutputDirectory }).minify({
  'path/to/source/1': {
    styles: '...styles...',
```

How to preserve a comment block?

Use the `/*!` notation instead of the standard one `/*`:

```
/*!
  Important comments included in
        optimized output.
*/
```

How to rebase relative image URLs?

clean-css will handle it automatically for you in the following cases:

- when full paths to input files are passed in as options;
- when correct paths are passed in via a hash;
- when `rebaseTo` is used with any of above two.

How to work with source maps?

To generate a source map, use `sourceMap: true` option, e.g.:


```
new CleanCSS({
  fetch: function (uri, inlineRequest,
         inlineTimeout, callback) {
    request(uri, function (error,
         response, body) {
      if (error) {
        callback(error, null);
      } else if (response &&
         response.statusCode != 200) {
        callback(response.statusCode,
        null);
      } else {
        callback(null, body);
      }
    });
  }
}).minify(source);
```

This option provides a convenient way of overriding the default fetching logic if it doesn't support a particular feature, say CONNECT proxies.

Unless given, the default loadRemoteResource logic is used.

Formatting options

By default output CSS is formatted without any whitespace unless a `format` option is given. First of all there are two shorthands:

```
new CleanCSS({
  format: 'beautify' // formats output
        in a really nice way
})
```

and

```
new CleanCSS({
  format: 'keep-breaks' // formats
        output the default way but adds
        line breaks for improved
        readability
})
```

however `format` option also accept a fine-grained set of options:

```
new CleanCSS({
  format: {
    breaks: { // controls where to
        insert breaks
      afterAtRule:
        false, // controls if a line
        break comes after an at-rule;
        e.g. `@charset`; defaults to
        `false`
      afterBlockBegins: false, //
        controls if a line break comes
        after a block begins; e.g.
        `@media`; defaults to `false`
      afterBlockEnds:
        false, // controls if a line
        break comes after a block ends,
        defaults to `false`
```



```
      }
    }
}).minify(source)
```

How to keep a CSS fragment intact?

Note: available since 4.2.0.

Wrap the CSS fragment in special comments which instruct clean-css to preserve it, e.g.

```
.block-1 {
  color: red
}
/* clean-css ignore:start */
.block-special {
  color: transparent
}
/* clean-css ignore:end */
.block-2 {
  margin: 0
}
```

Optimizing this CSS will result in the following output:

```
.block-1{color:red}
.block-special {
  color: transparent
}
.block-2{margin:0}
```

How to specify a custom rounding precision?

The level 1 `roundingPrecision` optimization option accept a string with per-unit rounding precision settings, e.g.

```
new CleanCSS({
  level: {
    1: {
      roundingPrecision: 'all=3,px=5'
    }
  }
}).minify(source)
```

which sets all units rounding precision to 3 digits except `px` unit precision of 5 digits.

How to optimize a stylesheet with custom `rpx` units?

Since `rpx` is a non standard unit (see #1074), it will be dropped by default as an invalid value.

However you can treat `rpx` units as regular ones:

```
new CleanCSS({
  compatibility: {
    customUnits: {
      rpx: true
```



```
    afterComment: false, // controls
      if a line break comes after a
      comment; defaults to `false`
    afterProperty: false, // controls
      if a line break comes after a
      property; defaults to `false`
    afterRuleBegins: false, //
      controls if a line break comes
      after a rule begins; defaults to
      `false`
    afterRuleEnds: false, // controls
      if a line break comes after a
      rule ends; defaults to `false`
    beforeBlockEnds: false, //
      controls if a line break comes
      before a block ends; defaults to
      `false`
    betweenSelectors: false //
      controls if a line break comes
      between selectors; defaults to
      `false`
  },
  breakWith:
      '\n', // controls the new line
      character, can be `'\r\n'` or
      `'\n'` (aliased as `'windows'`
      and `'unix'` or `'crlf'` and
      `'lf'`); defaults to system one,
      so former on Windows and latter
      on Unix
  indentBy: 0, // controls number of
      characters to indent with;
      defaults to `0`
```

```
    indentWith: 'space', // controls a
        character to indent with, can be
        ``'space'`` or ``'tab'``; defaults
        to ``'space'``
    spaces: { // controls where to
        insert spaces
      aroundSelectorRelation: false, //
        controls if spaces come around
        selector relations; e.g. `div >
        a`; defaults to `false`
      beforeBlockBegins: false, //
        controls if a space comes before
        a block begins; e.g. `.block {`;
        defaults to `false`
      beforeValue: false // controls if
        a space comes before a value;
        e.g. `width: 1rem`; defaults to
        `false`
    },
    wrapAt: false, // controls maximum
        line length; defaults to `false`
    semicolonAfterLastProperty:
        false // controls removing
        trailing semicolons in rule;
        defaults to `false` - means
        remove
  }
})
```

Also since clean-css 5.0 you can use numerical values for all line breaks, which will repeat a line break that many times, e.g:

```
  new CleanCSS({
    format: {
```


How to process remote `@imports` correctly?

In order to inline remote `@import` statements you need to provide a callback to minify method as fetching remote assets is an asynchronous operation, e.g.:

```
var source = '@import url(http://
        example.com/path/to/remote/
        styles);';
new CleanCSS({ inline:
        ['remote'] }).minify(source,
        function (error, output) {
  // output.styles
});
```

If you don't provide a callback, then remote `@imports` will be left as is.

How to apply arbitrary transformations to CSS properties?

Please see plugins.

use a single hash the order is determined by the traversal order of object properties - available since 4.1.0.

Important note - any `@import` rules already present in the hash will be resolved in memory.

How to process multiple files without concatenating them into one output file?

Since clean-css 5.0 you can, when passing an array of paths, hash, or array of hashes (see above), ask clean-css not to join styles into one output, but instead return stylesheets optimized one by one, e.g.

```
var output = new CleanCSS({ batch:
        true }).minify(['path/to/file/
        one', 'path/to/file/two']);
var outputOfFile1 = output['path/to/
        file/one'].styles // all other
        fields, like errors, warnings,
        or stats are there too
var outputOfFile2 = output['path/to/
        file/two'].styles
```



```
      breaks: {
        afterAtRule: 2,
        afterBlockBegins: 1, // 1 is
         synonymous with `true`
        afterBlockEnds: 2,
        afterComment: 1,
        afterProperty: 1,
        afterRuleBegins: 1,
        afterRuleEnds: 1,
        beforeBlockEnds: 1,
        betweenSelectors: 0 // 0 is
         synonymous with `false`
      }
    }
  })
```

which will add nicer spacing between at rules and blocks.

Inlining options

`inline` option whitelists which `@import` rules will be processed, e.g.

```
new CleanCSS({
  inline: ['local'] // default; enables
          local inlining only
})
```

```
new CleanCSS({
  inline: ['none'] // disables all
        inlining
})

// introduced in clean-css 4.1.0

new CleanCSS({
  inline:
        false // disables all inlining
        (alias to `['none']`)
})

new CleanCSS({
  inline: ['all'] // enables all
        inlining, same as ['local',
        'remote']
})

new CleanCSS({
  inline: ['local',
        'mydomain.example.com'] //
        enables local inlining plus
        given remote source
})

new CleanCSS({
  inline: ['local', 'remote', '!
        fonts.googleapis.com'] //
        enables all inlining but from
        given remote source
})
```


FAQ

How to optimize multiple files?

It can be done either by passing an array of paths, or, when sources are already available, a hash or an array of hashes:

```
new CleanCSS().minify(['path/to/file/
        one', 'path/to/file/two']);

new CleanCSS().minify({
  'path/to/file/one': {
    styles: 'contents of file one'
  },
  'path/to/file/two': {
    styles: 'contents of file two'
  }
});

new CleanCSS().minify([
  {'path/to/file/one': {styles:
        'contents of file one'}},
  {'path/to/file/two': {styles:
        'contents of file two'}}
]);
```

Passing an array of hashes allows you to explicitly specify the order in which the input files are concatenated. Whereas when you

CLI utility

Clean-css has an associated command line utility that can be installed separately using `npm install clean-css-cli`. For more detailed information, please visit https://github.com/clean-css/clean-css-cli.

Optimization levels

The `level` option can be either `0`, `1` (default), or `2`, e.g.

```
new CleanCSS({
  level: 2
})
```

or a fine-grained configuration given via a hash.

Please note that level 1 optimization options are generally safe while level 2 optimizations should be safe for most users.

Level 0 optimizations

Level 0 optimizations simply means “no optimizations”. Use it when you’d like to inline imports and / or rebase URLs but skip everything else.

Level 1 optimizations

Level 1 optimizations (default) operate on single properties only, e.g. can remove units when not required, turn rgb colors to a shorter hex representation, remove comments, etc

Here is a full list of available options:

```
new CleanCSS({
  level: {
```

```
1: {
  cleanupCharsets:
    true, // controls `@charset`
    moving to the front of a
    stylesheet; defaults to `true`
  normalizeUrls: true, // controls
    URL normalization; defaults to
    `true`
  optimizeBackground: true, //
    controls `background` property
    optimizations; defaults to
    `true`
  optimizeBorderRadius: true, //
    controls `border-radius`
    property optimizations; defaults
    to `true`
  optimizeFilter: true, // controls
    `filter` property optimizations;
    defaults to `true`
  optimizeFont: true, // controls
    `font` property optimizations;
    defaults to `true`
  optimizeFontWeight: true, //
    controls `font-weight` property
    optimizations; defaults to
    `true`
  optimizeOutline:
    true, // controls `outline`
    property optimizations; defaults
    to `true`
  removeEmpty: true, // controls
    removing empty rules and nested
    blocks; defaults to `true`
```



```
new CleanCSS(options).minify(source,
        function (error, output) {
  // `output` is the same as in the
        synchronous call above
});
```

To optimize a single file, without reading it first, pass a path to it to `minify` method as follows:

```
var output = new
        CleanCSS(options).minify(['path/
        to/file.css'])
```

(if you won't enclose the path in an array, it will be treated as a CSS source instead).

There are several ways to optimize multiple files at the same time, see How to optimize multiple files?.

Promise interface

If you prefer clean-css to return a Promise object then you need to explicitely ask for it, e.g.

```
new CleanCSS({ returnPromise: true })
  .minify(source)
  .then(function (output) {
        console.log(output.styles); })
  .catch(function (error)
        { // deal with errors });
```

```
  }
  div {
    margin: 5px
  }

`);

console.log(output);

// Log:
{
  styles: 'a{color:#00f}
        div{margin:5px}',
  stats: {
    efficiency: 0.6704545454545454,
    minifiedSize: 29,
    originalSize: 88,
    timeSpent: 6
  },
  errors: [],
  inlinedStylesheets: [],
  warnings: []
}
```

The `minify` method also accepts an input source map, e.g.

```
var output = new
        CleanCSS(options).minify(source,
        inputSourceMap);
```

or a callback invoked when optimizations are finished, e.g.


```
removeNegativePaddings: true, //
  controls removing negative
  paddings; defaults to `true`
removeQuotes: true, // controls
  removing quotes when
  unnecessary; defaults to `true`
removeWhitespace: true, //
  controls removing unused
  whitespace; defaults to `true`
replaceMultipleZeros: true, //
  contols removing redundant
  zeros; defaults to `true`
replaceTimeUnits: true, //
  controls replacing time units
  with shorter values; defaults to
  `true`
replaceZeroUnits: true, //
  controls replacing zero values
  with units; defaults to `true`
roundingPrecision: false, //
  rounds pixel values to `N`
  decimal places; `false` disables
  rounding; defaults to `false`
selectorsSortingMethod:
  'standard', // denotes selector
  sorting method; can be
  ``'natural'`` or ``'standard'``,
  ``'none'``, or false (the last two
  since 4.1.0); defaults to
  ``'standard'``
specialComments:
  'all', // denotes a number of /
  *! ... */ comments preserved;
  defaults to `all`
```

```
      tidyAtRules: true, // controls at-
        rules (e.g. `@charset`,
        `@import`) optimizing; defaults
        to `true`
      tidyBlockScopes:
        true, // controls block scopes
        (e.g. `@media`) optimizing;
        defaults to `true`
      tidySelectors: true, // controls
        selectors optimizing; defaults
        to `true`,
      variableValueOptimizers: [] //
        controls value optimizers which
        are applied to variables
    }
  }
});
```

There is an `all` shortcut for toggling all options at the same time, e.g.

```
new CleanCSS({
  level: {
    1: {
      all: false, // set all values to
        `false`
      tidySelectors: true // turns on
        optimizing selectors
    }
  }
});
```


The output of the `minify` method is a hash with following fields:

```
console.log(output.styles); // optimized
        output CSS as a string
console.log(output.sourceMap); // output
        source map if requested with
        `sourceMap` option
console.log(output.errors); // a list of
        errors raised
console.log(output.warnings); // a list
        of warnings raised
console.log(output.stats.originalSize); //
        original content size after
        import inlining
console.log(output.stats.minifiedSize); //
        optimized content size
console.log(output.stats.timeSpent); //
        time spent on optimizations in
        milliseconds
console.log(output.stats.efficiency); //
        `(originalSize - minifiedSize) /
        originalSize`, e.g. 0.25 if size
        is reduced from 100 bytes to 75
        bytes
```

Example: Minifying a CSS string:

```
const CleanCSS = require("clean-css");

const output = new CleanCSS().minify(`

  a {
    color: blue;
```

```
      if (property.name == 'background-
        repeat' &&
        property.value.length == 2 &&
        property.value[0][1] ==
        property.value[1][1]) {
        property.value.pop();
        property.dirty = true;
      }
    }
  }
}

new CleanCSS({plugins: [myPlugin]})
```

Search `test\module-test.js` for `plugins` or check out `lib/optimizer/level-1/property-optimizers` and `lib/optimizer/level-1/value-optimizers` for more examples.

Important: To rewrite your old `transform` as a plugin, check out this commit.

Minify method

Once configured clean-css provides a `minify` method to optimize a given CSS, e.g.

```
var output = new
        CleanCSS(options).minify(source);
```


Level 2 optimizations

Level 2 optimizations operate at rules or multiple properties level, e.g. can remove duplicate rules, remove properties redefined further down a stylesheet, or restructure rules by moving them around.

Please note that if level 2 optimizations are turned on then, unless explicitely disabled, level 1 optimizations are applied as well.

Here is a full list of available options:

```
new CleanCSS({
  level: {
    2: {
      mergeAdjacentRules: true, //
        controls adjacent rules merging;
        defaults to true
      mergeIntoShorthands: true, //
        controls merging properties into
        shorthands; defaults to true
      mergeMedia: true, // controls
        `@media` merging; defaults to
        true
      mergeNonAdjacentRules: true, //
        controls non-adjacent rule
        merging; defaults to true
      mergeSemantically: false, //
        controls semantic merging;
        defaults to false
```

```
  overrideProperties: true, //
    controls property overriding
    based on understandability;
    defaults to true
  removeEmpty: true, // controls
    removing empty rules and nested
    blocks; defaults to `true`
  reduceNonAdjacentRules: true, //
    controls non-adjacent rule
    reducing; defaults to true
  removeDuplicateFontRules:
    true, // controls duplicate
    `@font-face` removing; defaults
    to true
  removeDuplicateMediaBlocks:
    true, // controls duplicate
    `@media` removing; defaults to
    true
  removeDuplicateRules: true, //
    controls duplicate rules
    removing; defaults to true
  removeUnusedAtRules: false, //
    controls unused at rule
    removing; defaults to false
    (available since 4.1.0)
  restructureRules: false, //
    controls rule restructuring;
    defaults to false
  skipProperties: [] // controls
    which properties won't be
    optimized, defaults to `[]`
    which means all will be
    optimized (since 4.1.0)
}
```



```
  }
});
```

There is an `all` shortcut for toggling all options at the same time, e.g.

```
new CleanCSS({
  level: {
    2: {
      all: false, // sets all values to
        `false`
      removeDuplicateRules: true //
        turns on removing duplicate
        rules
    }
  }
});
```

Plugins

In clean-css version 5 and above you can define plugins which run alongside level 1 and level 2 optimizations, e.g.

```
var myPlugin = {
  level1: {
    property: function
        removeRepeatedBackgroundRepeat(_rule,
        property, _options) {
      // So `background-repeat:no-repeat
        no-repeat` becomes `background-
        repeat:no-repeat`
```