# API Documentation

*Please use only this documented API when working with the parser. Methods not documented here are subject to change at any point.*

## parser function

This is the module's main entry point.

const parser = require('postcss-selector-parser');

### parser([transform], [options])

Creates a new processor instance

const processor = parser();

Or, with optional transform function

const transform = selectors => {

selectors.walkUniversals(selector => {

selector.remove();

});

};

const processor = parser(transform)

// Example

const result = processor.processSync('\*.class');

// => .class

[See processor documentation](#processor)

Arguments:

* transform (function): Provide a function to work with the parsed AST.
* options (object): Provide default options for all calls on the returned Processor.

### parser.attribute([props])

Creates a new attribute selector.

parser.attribute({attribute: 'href'});

// => [href]

Arguments:

* props (object): The new node's properties.

### parser.className([props])

Creates a new class selector.

parser.className({value: 'button'});

// => .button

Arguments:

* props (object): The new node's properties.

### parser.combinator([props])

Creates a new selector combinator.

parser.combinator({value: '+'});

// => +

Arguments:

* props (object): The new node's properties.

Notes:

* **Descendant Combinators** The value of descendant combinators created by the parser always just a single space (" "). For descendant selectors with no comments, additional space is now stored in node.spaces.before. Depending on the location of comments, additional spaces may be stored in node.raws.spaces.before, node.raws.spaces.after, or node.raws.value.
* **Named Combinators** Although, nonstandard and unlikely to ever become a standard, named combinators like /deep/ and /for/ are parsed as combinators. The node.value is name after being unescaped and normalized as lowercase. The original value for the combinator name is stored in node.raws.value.

### parser.comment([props])

Creates a new comment.

parser.comment({value: '/\* Affirmative, Dave. I read you. \*/'});

// => /\* Affirmative, Dave. I read you. \*/

Arguments:

* props (object): The new node's properties.

### parser.id([props])

Creates a new id selector.

parser.id({value: 'search'});

// => #search

Arguments:

* props (object): The new node's properties.

### parser.nesting([props])

Creates a new nesting selector.

parser.nesting();

// => &

Arguments:

* props (object): The new node's properties.

### parser.pseudo([props])

Creates a new pseudo selector.

parser.pseudo({value: '::before'});

// => ::before

Arguments:

* props (object): The new node's properties.

### parser.root([props])

Creates a new root node.

parser.root();

// => (empty)

Arguments:

* props (object): The new node's properties.

### parser.selector([props])

Creates a new selector node.

parser.selector();

// => (empty)

Arguments:

* props (object): The new node's properties.

### parser.string([props])

Creates a new string node.

parser.string();

// => (empty)

Arguments:

* props (object): The new node's properties.

### parser.tag([props])

Creates a new tag selector.

parser.tag({value: 'button'});

// => button

Arguments:

* props (object): The new node's properties.

### parser.universal([props])

Creates a new universal selector.

parser.universal();

// => \*

Arguments:

* props (object): The new node's properties.

## Node types

### node.type

A string representation of the selector type. It can be one of the following; attribute, class, combinator, comment, id, nesting, pseudo, root, selector, string, tag, or universal. Note that for convenience, these constants are exposed on the main parser as uppercased keys. So for example you can get id by querying parser.ID.

parser.attribute({attribute: 'href'}).type;

// => 'attribute'

### node.parent

Returns the parent node.

root.nodes[0].parent === root;

### node.toString(), String(node), or '' + node

Returns a string representation of the node.

const id = parser.id({value: 'search'});

console.log(String(id));

// => #search

### node.next() & node.prev()

Returns the next/previous child of the parent node.

const next = id.next();

if (next && next.type !== 'combinator') {

throw new Error('Qualified IDs are not allowed!');

}

### node.replaceWith(node)

Replace a node with another.

const attr = selectors.first.first;

const className = parser.className({value: 'test'});

attr.replaceWith(className);

Arguments:

* node: The node to substitute the original with.

### node.remove()

Removes the node from its parent node.

if (node.type === 'id') {

node.remove();

}

### node.clone([opts])

Returns a copy of a node, detached from any parent containers that the original might have had.

const cloned = node.clone();

### node.isAtPosition(line, column)

Return a boolean indicating whether this node includes the character at the position of the given line and column. Returns undefined if the nodes lack sufficient source metadata to determine the position.

Arguments:

* line: 1-index based line number relative to the start of the selector.
* column: 1-index based column number relative to the start of the selector.

### node.spaces

Extra whitespaces around the node will be moved into node.spaces.before and node.spaces.after. So for example, these spaces will be moved as they have no semantic meaning:

h1 , h2 {}

For descendent selectors, the value is always a single space.

h1 h2 {}

Additional whitespace is found in either the node.spaces.before and node.spaces.after depending on the presence of comments or other whitespace characters. If the actual whitespace does not start or end with a single space, the node's raw value is set to the actual space(s) found in the source.

### node.source

An object describing the node's start/end, line/column source position.

Within the following CSS, the .bar class node ...

.foo,

.bar {}

... will contain the following source object.

source: {

start: {

line: 2,

column: 3

},

end: {

line: 2,

column: 6

}

}

### node.sourceIndex

The zero-based index of the node within the original source string.

Within the following CSS, the .baz class node will have a sourceIndex of 12.

.foo, .bar, .baz {}

## Container types

The root, selector, and pseudo nodes have some helper methods for working with their children.

### container.nodes

An array of the container's children.

// Input: h1 h2

selectors.at(0).nodes.length // => 3

selectors.at(0).nodes[0].value // => 'h1'

selectors.at(0).nodes[1].value // => ' '

### container.first & container.last

The first/last child of the container.

selector.first === selector.nodes[0];

selector.last === selector.nodes[selector.nodes.length - 1];

### container.at(index)

Returns the node at position index.

selector.at(0) === selector.first;

selector.at(0) === selector.nodes[0];

Arguments:

* index: The index of the node to return.

### container.atPosition(line, column)

Returns the node at the source position line and column.

// Input: :not(.foo),\n#foo > :matches(ol, ul)

selector.atPosition(1, 1); // => :not(.foo)

selector.atPosition(2, 1); // => \n#foo

Arguments:

* line: The line number of the node to return.
* column: The column number of the node to return.

### container.index(node)

Return the index of the node within its container.

selector.index(selector.nodes[2]) // => 2

Arguments:

* node: A node within the current container.

### container.length

Proxy to the length of the container's nodes.

container.length === container.nodes.length

### container Array iterators

The container class provides proxies to certain Array methods; these are:

* container.map === container.nodes.map
* container.reduce === container.nodes.reduce
* container.every === container.nodes.every
* container.some === container.nodes.some
* container.filter === container.nodes.filter
* container.sort === container.nodes.sort

Note that these methods only work on a container's immediate children; recursive iteration is provided by container.walk.

### container.each(callback)

Iterate the container's immediate children, calling callback for each child. You may return false within the callback to break the iteration.

let className;

selectors.each((selector, index) => {

if (selector.type === 'class') {

className = selector.value;

return false;

}

});

Note that unlike Array#forEach(), this iterator is safe to use whilst adding or removing nodes from the container.

Arguments:

* callback (function): A function to call for each node, which receives node and index arguments.

### container.walk(callback)

Like container#each, but will also iterate child nodes as long as they are container types.

selectors.walk((selector, index) => {

// all nodes

});

Arguments:

* callback (function): A function to call for each node, which receives node and index arguments.

This iterator is safe to use whilst mutating container.nodes, like container#each.

### container.walk proxies

The container class provides proxy methods for iterating over types of nodes, so that it is easier to write modules that target specific selectors. Those methods are:

* container.walkAttributes
* container.walkClasses
* container.walkCombinators
* container.walkComments
* container.walkIds
* container.walkNesting
* container.walkPseudos
* container.walkTags
* container.walkUniversals

### container.split(callback)

This method allows you to split a group of nodes by returning true from a callback. It returns an array of arrays, where each inner array corresponds to the groups that you created via the callback.

// (input) => h1 h2>>h3

const list = selectors.first.split(selector => {

return selector.type === 'combinator';

});

// (node values) => [['h1', ' '], ['h2', '>>'], ['h3']]

Arguments:

* callback (function): A function to call for each node, which receives node as an argument.

### container.prepend(node) & container.append(node)

Add a node to the start/end of the container. Note that doing so will set the parent property of the node to this container.

const id = parser.id({value: 'search'});

selector.append(id);

Arguments:

* node: The node to add.

### container.insertBefore(old, new) & container.insertAfter(old, new)

Add a node before or after an existing node in a container:

selectors.walk(selector => {

if (selector.type !== 'class') {

const className = parser.className({value: 'theme-name'});

selector.parent.insertAfter(selector, className);

}

});

Arguments:

* old: The existing node in the container.
* new: The new node to add before/after the existing node.

### container.removeChild(node)

Remove the node from the container. Note that you can also use node.remove() if you would like to remove just a single node.

selector.length // => 2

selector.remove(id)

selector.length // => 1;

id.parent // undefined

Arguments:

* node: The node to remove.

### container.removeAll() or container.empty()

Remove all children from the container.

selector.removeAll();

selector.length // => 0

## Root nodes

A root node represents a comma separated list of selectors. Indeed, all a root's toString() method does is join its selector children with a ','. Other than this, it has no special functionality and acts like a container.

### root.trailingComma

This will be set to true if the input has a trailing comma, in order to support parsing of legacy CSS hacks.

## Selector nodes

A selector node represents a single complex selector. For example, this selector string h1 h2 h3, [href] > p, is represented as two selector nodes. It has no special functionality of its own.

## Pseudo nodes

A pseudo selector extends a container node; if it has any parameters of its own (such as h1:not(h2, h3)), they will be its children. Note that the pseudo value will always contain the colons preceding the pseudo identifier. This is so that both :before and ::before are properly represented in the AST.

## Attribute nodes

### attribute.quoted

Returns true if the attribute's value is wrapped in quotation marks, false if it is not. Remains undefined if there is no attribute value.

[href=foo] /\* false \*/

[href='foo'] /\* true \*/

[href="foo"] /\* true \*/

[href] /\* undefined \*/

### attribute.qualifiedAttribute

Returns the attribute name qualified with the namespace if one is given.

### attribute.offsetOf(part)

Returns the offset of the attribute part specified relative to the start of the node of the output string. This is useful in raising error messages about a specific part of the attribute, especially in combination with attribute.sourceIndex.

Returns -1 if the name is invalid or the value doesn't exist in this attribute.

The legal values for part are:

* "ns" - alias for "namespace"
* "namespace" - the namespace if it exists.
* "attribute" - the attribute name
* "attributeNS" - the start of the attribute or its namespace
* "operator" - the match operator of the attribute
* "value" - The value (string or identifier)
* "insensitive" - the case insensitivity flag

### attribute.raws.unquoted

Returns the unquoted content of the attribute's value. Remains undefined if there is no attribute value.

[href=foo] /\* foo \*/

[href='foo'] /\* foo \*/

[href="foo"] /\* foo \*/

[href] /\* undefined \*/

### attribute.spaces

Like node.spaces with the before and after values containing the spaces around the element, the parts of the attribute can also have spaces before and after them. The for each of attribute, operator, value and insensitive there is corresponding property of the same nam in node.spaces that has an optional before or after string containing only whitespace.

Note that corresponding values in attributes.raws.spaces contain values including any comments. If set, these values will override the attribute.spaces value. Take care to remove them if changing attribute.spaces.

### attribute.raws

The raws object stores comments and other information necessary to re-render the node exactly as it was in the source.

If a comment is embedded within the identifiers for the namespace, attribute or value then a property is placed in the raws for that value containing the full source of the propery including comments.

If a comment is embedded within the space between parts of the attribute then the raw for that space is set accordingly.

Setting an attribute's property raws value to be deleted.

For now, changing the spaces required also updating or removing any of the raws values that override them.

Example: [ /\*before\*/ href /\* after-attr \*/ = /\* after-operator \*/ te/\*inside-value\*/st/\* wow \*/ /\*omg\*/i/\*bbq\*/ /\*whodoesthis\*/] would parse as:

{

attribute: "href",

operator: "=",

value: "test",

spaces: {

before: '',

after: '',

attribute: { before: ' ', after: ' ' },

operator: { after: ' ' },

value: { after: ' ' },

insensitive: { after: ' ' }

},

raws: {

spaces: {

attribute: { before: ' /\*before\*/ ', after: ' /\* after-attr \*/ ' },

operator: { after: ' /\* after-operator \*/ ' },

value: { after: '/\* wow \*/ /\*omg\*/' },

insensitive: { after: '/\*bbq\*/ /\*whodoesthis\*/' }

},

unquoted: 'test',

value: 'te/\*inside-value\*/st'

}

}

## Processor

### ProcessorOptions

* lossless - When true, whitespace is preserved. Defaults to true.
* updateSelector - When true, if any processor methods are passed a postcss Rule node instead of a string, then that Rule's selector is updated with the results of the processing. Defaults to true.

### process|processSync(selectors, [options])

Processes the selectors, returning a string from the result of processing.

Note: when the updateSelector option is set, the rule's selector will be updated with the resulting string.

**Example:**

const parser = require("postcss-selector-parser");

const processor = parser();

let result = processor.processSync(' .class');

console.log(result);

// => .class

// Asynchronous operation

let promise = processor.process(' .class').then(result => {

console.log(result)

// => .class

});

// To have the parser normalize whitespace values, utilize the options

result = processor.processSync(' .class ', {lossless: false});

console.log(result);

// => .class

// For better syntax errors, pass a PostCSS Rule node.

const postcss = require('postcss');

rule = postcss.rule({selector: ' #foo > a, .class '});

processor.process(rule, {lossless: false, updateSelector: true}).then(result => {

console.log(result);

// => #foo>a,.class

console.log("rule:", rule.selector);

// => rule: #foo>a,.class

})

Arguments:

* selectors (string|postcss.Rule): Either a selector string or a PostCSS Rule node.
* [options] (object): Process options

### ast|astSync(selectors, [options])

Like process() and processSync() but after processing the selectors these methods return the Root node of the result instead of a string.

Note: when the updateSelector option is set, the rule's selector will be updated with the resulting string.

### transform|transformSync(selectors, [options])

Like process() and processSync() but after processing the selectors these methods return the value returned by the processor callback.

Note: when the updateSelector option is set, the rule's selector will be updated with the resulting string.

### Error Handling Within Selector Processors

The root node passed to the selector processor callback has a method error(message, options) that returns an error object. This method should always be used to raise errors relating to the syntax of selectors. The options to this method are passed to postcss's error constructor ([documentation](http://postcss.org/api/#container-error)).

#### Async Error Example

let processor = (root) => {

return new Promise((resolve, reject) => {

root.walkClasses((classNode) => {

if (/^(.\*)[-\_]/.test(classNode.value)) {

let msg = "classes may not have underscores or dashes in them";

reject(root.error(msg, {

index: classNode.sourceIndex + RegExp.$1.length + 1,

word: classNode.value

}));

}

});

resolve();

});

};

const postcss = require("postcss");

const parser = require("postcss-selector-parser");

const selectorProcessor = parser(processor);

const plugin = postcss.plugin('classValidator', (options) => {

return (root) => {

let promises = [];

root.walkRules(rule => {

promises.push(selectorProcessor.process(rule));

});

return Promise.all(promises);

};

});

postcss(plugin()).process(`

.foo-bar {

color: red;

}

`.trim(), {from: 'test.css'}).catch((e) => console.error(e.toString()));

// CssSyntaxError: classValidator: ./test.css:1:5: classes may not have underscores or dashes in them

//

// > 1 | .foo-bar {

// | ^

// 2 | color: red;

// 3 | }

#### Synchronous Error Example

let processor = (root) => {

root.walkClasses((classNode) => {

if (/.\*[-\_]/.test(classNode.value)) {

let msg = "classes may not have underscores or dashes in them";

throw root.error(msg, {

index: classNode.sourceIndex,

word: classNode.value

});

}

});

};

const postcss = require("postcss");

const parser = require("postcss-selector-parser");

const selectorProcessor = parser(processor);

const plugin = postcss.plugin('classValidator', (options) => {

return (root) => {

root.walkRules(rule => {

selectorProcessor.processSync(rule);

});

};

});

postcss(plugin()).process(`

.foo-bar {

color: red;

}

`.trim(), {from: 'test.css'}).catch((e) => console.error(e.toString()));

// CssSyntaxError: classValidator: ./test.css:1:5: classes may not have underscores or dashes in them

//

// > 1 | .foo-bar {

// | ^

// 2 | color: red;

// 3 | }