# Marked Documentation



## Extending Marked

To champion the single-responsibility and open/closed principles, we have tried to make it relatively painless to extend Marked. If you are looking to add custom functionality, this is the place to start.

## marked.use()

`marked.use(extension)` is the recommended way to extend Marked. The `extension` object can contain any option available in Marked:

```
import { marked } from 'marked';

marked.use({
  pedantic: false,
  gfm: true,
  breaks: false,
  sanitize: false,
  smartypants: false,
  xhtml: false
});
```

You can also supply multiple `extension` objects at once.

```
marked.use(myExtension, extension2, extension3);

\\ EQUIVALENT TO:

marked.use(myExtension);
marked.use(extension2);
marked.use(extension3);
```

All options will overwrite those previously set, except for the following options which will be merged with the existing framework and can be used to change or extend the functionality of Marked: `renderer`, `tokenizer`, `walkTokens`, and `extensions`.

- The `renderer` and `tokenizer` options are objects with functions that will be merged into the built-in `renderer` and `tokenizer` respectively.
- The `walkTokens` option is a function that will be called to post-process every token before rendering.
- The `extensions` option is an array of objects that can contain additional custom `renderer` and `tokenizer` steps that will execute before any of the default parsing logic occurs.

---

## The Marked Pipeline

Before building your custom extensions, it is important to understand the components that Marked uses to translate from Markdown to HTML:

1. The user supplies Marked with an input string to be translated.
2. The `lexer` feeds segments of the input text string into each `tokenizer`, and from their output, generates a series of tokens in a nested tree structure.
3. Each `tokenizer` receives a segment of Markdown text and, if it matches a particular pattern, generates a token object containing any relevant information.
4. The `walkTokens` function will traverse every token in the tree and perform any final adjustments to the token contents.
5. The `parser` traverses the token tree and feeds each token into the appropriate `renderer`, and concatenates their outputs into the final HTML result.
6. Each `renderer` receives a token and manipulates its contents to generate a segment of HTML.

Marked provides methods for directly overriding the `renderer` and `tokenizer` for any existing token type, as well as inserting additional custom `renderer` and `tokenizer` functions to handle entirely custom syntax.

---

## The Renderer : `renderer`

The renderer defines the HTML output of a given token. If you supply a `renderer` in the options object passed to `marked.use()`, any functions in the object will override the default handling of that token type.

Calling `marked.use()` to override the same function multiple times will give priority to the version that was assigned *last*. Overriding functions can return `false` to fall back to the previous override in the sequence, or resume default behavior if all overrides return `false`. Returning any other value (including nothing) will prevent fallback behavior.

**Example:** Overriding output of the default `heading` token by adding an embedded anchor tag like on GitHub.

```
// Create reference instance
import { marked } from 'marked';

// Override function
const renderer = {
  heading(text, level) {
    const escapedText = text.toLowerCase().replace(/[^\w]+/g, '-');

    return `
            <h${level}>
              <a name="${escapedText}" class="anchor" href="#${escapedText}">
                <span class="header-link"></span>
              </a>
              ${text}
            </h${level}>`;
  }
};

marked.use({ renderer });

// Run marked
console.log(marked.parse('# heading+'));
```

**Output:**

```
<h1>
  <a name="heading-" class="anchor" href="#heading-">
    <span class="header-link"></span>
  </a>
  heading+
</h1>
```

### Block-level renderer methods

- `code(`*`string`* `code, string infostring, boolean escaped)`
- `blockquote(`*`string`* `quote)`
- `html(`*`string`* `html, boolean block)`

- `heading(string text, number level, string raw, Slugger slugger)`
- `hr()`
- `list(string body, boolean ordered, number start)`
- `listitem(string text, boolean task, boolean checked)`
- `checkbox(boolean checked)`
- `paragraph(string text)`
- `table(string header, string body)`
- `tablerow(string content)`
- `tablecell(string content, object flags)`

#### Inline-level renderer methods

- `strong(string text)`
- `em(string text)`
- `codespan(string code)`
- `br()`
- `del(string text)`
- `link(string href, string title, string text)`
- `image(string href, string title, string text)`
- `text(string text)`

`Slugger` is exposed from marked as `marked.Slugger`:

```
import { marked } from 'marked'
const slugger = new marked.Slugger()
```

`slugger` has the `slug` method to create a unique id from value:

```
slugger.slug('foo')   // foo
slugger.slug('foo')   // foo-1
slugger.slug('foo')   // foo-2
slugger.slug('foo 1') // foo-1-1
slugger.slug('foo-1') // foo-1-2
...
```

`slugger.slug` can also be called with the `dryrun` option for stateless operation:

```
slugger.slug('foo')                   // foo
slugger.slug('foo')                   // foo-1
slugger.slug('foo')                   // foo-2
slugger.slug('foo', { dryrun: true }) // foo-3
slugger.slug('foo', { dryrun: true }) // foo-3
slugger.slug('foo')                   // foo-3
slugger.slug('foo')                   // foo-4
...
```

`flags` has the following properties:

```
{
    header: true || false,
    align: 'center' || 'left' || 'right'
}
```

---

## The Tokenizer : `tokenizer`

The tokenizer defines how to turn markdown text into tokens. If you supply a `tokenizer` object to the Marked options, it will be merged with the built-in tokenizer and any functions inside will override the default handling of that token type.

Calling `marked.use()` to override the same function multiple times will give priority to the version that was assigned *last*. Overriding functions can return `false` to fall back to the previous override in the sequence, or resume default behavior if all overrides return `false`. Returning any other value (including nothing) will prevent fallback behavior.

**Example:** Overriding default `codespan` tokenizer to include LaTeX.

```
// Create reference instance
import { marked } from 'marked';

// Override function
const tokenizer = {
  codespan(src) {
    const match = src.match(/^\$+([^\$\n]+?)\$+/);
    if (match) {
      return {
        type: 'codespan',
        raw: match[0],
        text: match[1].trim()
      };
    }

    // return false to use original codespan tokenizer
    return false;
  }
};

marked.use({ tokenizer });

// Run marked
console.log(marked.parse('$ latex code $\n\n` other code `'));
```

**Output:**

```
<p><code>latex code</code></p>
<p><code>other code</code></p>
```

**NOTE**: This does not fully support latex, see issue #1948.

#### Block level tokenizer methods

- `space(string src)`
- `code(string src)`
- `fences(string src)`
- `heading(string src)`
- `hr(string src)`
- `blockquote(string src)`
- `list(string src)`
- `html(string src)`
- `def(string src)`
- `table(string src)`
- `lheading(string src)`
- `paragraph(string src)`
- `text(string src)`

#### Inline level tokenizer methods

- `escape(string src)`
- `tag(string src)`
- `link(string src)`
- `reflink(string src, object links)`
- `emStrong(string src, string maskedSrc, string prevChar)`
- `codespan(string src)`
- `br(string src)`
- `del(string src)`
- `autolink(string src, function mangle)`
- `url(string src, function mangle)`
- `inlineText(string src, function smartypants)`

`mangle` is a method that changes text to HTML character references:

```
mangle('test@example.com')
// "&#x74;&#101;&#x73;&#116;&#x40;&#101;&#120;&#x61;&#x6d;&#112;&#108;&#101;&#46;&#x63;&#111;&#x6d;"
```

`smartypants` is a method that translates plain ASCII punctuation characters into "smart" typographic punctuation HTML entities:

https://daringfireball.net/projects/smartypants/

```
smartypants('"this ... string"')
// ""this … string""
```

---

## Walk Tokens : `walkTokens`

The walkTokens function gets called with every token. Child tokens are called before moving on to sibling tokens. Each token is passed by reference so updates are persisted when passed to the parser. When `async` mode is enabled, the return value is awaited. Otherwise the return value is ignored.

`marked.use()` can be called multiple times with different `walkTokens` functions. Each function will be called in order, starting with the function that was assigned *last*.

**Example:** Overriding heading tokens to start at h2.

```
import { marked } from 'marked';

// Override function
const walkTokens = (token) => {
  if (token.type === 'heading') {
    token.depth += 1;
  }
};

marked.use({ walkTokens });

// Run marked
console.log(marked.parse('# heading 2\n\n## heading 3'));
```

**Output:**

```
<h2 id="heading-2">heading 2</h2>
<h3 id="heading-3">heading 3</h3>
```

---

## Hooks : `hooks`

Hooks are methods that hook into some part of marked. The following hooks are available:

| signature | description |
|---|---|
| `preprocess(markdown: string): string` | Process markdown before sending it to marked. |
| `postprocess(html: string): string` | Process html after marked has finished parsing. |

`marked.use()` can be called multiple times with different `hooks` functions. Each function will be called in order, starting with the function that was assigned *last*.

**Example:** Set options based on front-matter

```
import { marked } from 'marked';
import fm from 'front-matter';

// Override function
const hooks = {
  preprocess(markdown) {
    const { attributes, body } = fm(markdown);
    for (const prop in attributes) {
      if (prop in this.options) {
        this.options[prop] = attributes[prop];
      }
    }
    return body;
  }
};

marked.use({ hooks });

// Run marked
console.log(marked.parse(`
---
headerIds: false
---

## test
`.trim()));
```

**Output:**

```
<h2>test</h2>
```

**Example:** Sanitize HTML with isomorphic-dompurify

```
import { marked } from 'marked';
import DOMPurify from 'isomorphic-dompurify';

// Override function
const hooks = {
  postprocess(html) {
    return DOMPurify.sanitize(html);
  }
};

marked.use({ hooks });

// Run marked
```

```
console.log(marked.parse(
<img src=x onerror=alert(1)//>
`));
```

**Output:**

```
<img src="x">
```

---

## Custom Extensions : `extensions`

You may supply an `extensions` array to the `options` object. This array can contain any number of `extension` objects, using the following properties:

`name`
A string used to identify the token that will be handled by this extension.

If the name matches an existing extension name, or an existing method in the tokenizer/renderer methods listed above, they will override the previously assigned behavior, with priority on the extension that was assigned **last**. An extension can return `false` to fall back to the previous behavior.

`level`
A string to determine when to run the extension tokenizer. Must be equal to 'block' or 'inline'.

A **block-level** extension will be handled before any of the block-level tokenizer methods listed above, and generally consists of 'container-type' text (paragraphs, tables, blockquotes, etc.).

An **inline-level** extension will be handled inside each block-level token, before any of the inline-level tokenizer methods listed above. These generally consist of 'style-type' text (italics, bold, etc.).

`start(string src)`
A function that returns the index of the next potential start of the custom token.

The index can be the result of a `src.match().index`, or even a simple `src.index()`. Marked will use this function to ensure that it does not skip over any text that should be part of the custom token.

`tokenizer(string src, array tokens)`
A function that reads string of Markdown text and returns a generated token. The token pattern should be found at the beginning of the `src` string. Accordingly, if using a Regular Expression to detect a token, it should be anchored to the string start (`^`). The `tokens` parameter contains the array of tokens that have been generated by the lexer up to that point, and can be used to access the previous token, for instance.

The return value should be an object with the following parameters:

`type`
A string that matches the `name` parameter of the extension.
`raw`
A string containing all of the text that this token consumes from the source.
`tokens [optional]`
An array of child tokens that will be traversed by the `walkTokens` function by default.

The returned token can also contain any other custom parameters of your choice that your custom `renderer` might need to access.

The tokenizer function has access to the lexer in the `this` object, which can be used if any internal section of the string needs to be parsed further, such as in handling any inline syntax on the text within a block token. The key functions that may be useful include:

`this.lexer.blockTokens(string text, array tokens)`
This runs the block tokenizer functions (including any block-level extensions) on the provided text, and appends any resulting tokens onto the `tokens` array. The `tokens` array is also returned by the function. You might use this, for example, if your extension creates a "container"-type token (such as a blockquote) that can potentially include other block-level tokens inside.
`this.lexer.inline(string text, array tokens)`
Parsing of inline-level tokens only occurs after all block-level tokens have been generated. This function adds `text` and `tokens` to a queue to be processed using inline-level tokenizers (including any inline-level extensions) at that later step. Tokens will be generated using the provided `text`, and any resulting tokens will be appended to the `tokens` array. Note that this function does **NOT** return anything since the inline processing cannot happen until the block-level processing is complete.
`this.lexer.inlineTokens(string text, array tokens)`
Sometimes an inline-level token contains further nested inline tokens (such as a

```
**strong**
```

token inside of a

```
### Heading
```

). This runs the inline tokenizer functions (including any inline-level extensions) on the provided text, and appends any resulting tokens onto the `tokens` array. The `tokens` array is also returned by the function.

`renderer(object token)`
A function that reads a token and returns the generated HTML output string.

The renderer function has access to the parser in the `this` object, which can be used if any part of the token needs needs to be parsed further, such as any child tokens. The key functions that may be useful include:

`this.parser.parse(array tokens)`
Runs the block renderer functions (including any extensions) on the provided array of tokens, and returns the resulting HTML string output. This is used to generate the HTML from any child block-level tokens, for example if your extension is a "container"-type token (such as a blockquote) that can potentially include other block-level tokens inside.
`this.parser.parseInline(array tokens)`
Runs the inline renderer functions (including any extensions) on the provided array of tokens, and returns the resulting HTML string output. This is used to generate the HTML from any child inline-level tokens.

`childTokens [optional]`
An array of strings that match the names of any token parameters that should be traversed by the `walkTokens` functions. For instance, if you want to use a second custom parameter to contain child tokens in addition to `tokens`, it could be listed here. If `childTokens` is provided, the `tokens` array will not be walked by default unless it is also included in the `childTokens` array.

**Example:** Add a custom syntax to generate `<dl>` description lists.

```
const descriptionList = {
  name: 'descriptionList',
  level: 'block',                                     // Is this a block-level or inline-level tokenizer?
  start(src) { return src.match(/:[^:\n]/)?.index; }, // Hint to Marked.js to stop and check for a match
  tokenizer(src, tokens) {
    const rule = /^(?::[^:\n]+:[^:\n]*(?:\n|$))+/;    // Regex for the complete token, anchor to string start
    const match = rule.exec(src);
    if (match) {
      const token = {                                 // Token to generate
        type: 'descriptionList',                      // Should match "name" above
        raw: match[0],                                // Text to consume from the source
```

```
        text: match[0].trim(),                          // Additional custom properties
        tokens: []                                      // Array where child inline tokens will be generated
      };
      this.lexer.inline(token.text, token.tokens);    // Queue this data to be processed for inline tokens
      return token;
    }
  },
  renderer(token) {
    return `<dl>${this.parser.parseInline(token.tokens)}\n</dl>`; // parseInline to turn child tokens into HTML
  }
};

const description = {
  name: 'description',
  level: 'inline',                                  // Is this a block-level or inline-level tokenizer?
  start(src) { return src.match(/:/)?.index; },    // Hint to Marked.js to stop and check for a match
  tokenizer(src, tokens) {
    const rule = /^:([^:\n]+):([^:\n]*)(?:\n|$)/;  // Regex for the complete token, anchor to string start
    const match = rule.exec(src);
    if (match) {
      return {                                      // Token to generate
        type: 'description',                        // Should match "name" above
        raw: match[0],                              // Text to consume from the source
        dt: this.lexer.inlineTokens(match[1].trim()),  // Additional custom properties, including
        dd: this.lexer.inlineTokens(match[2].trim())   //   any further-nested inline tokens
      };
    }
  },
  renderer(token) {
    return `\n<dt>${this.parser.parseInline(token.dt)}</dt><dd>${this.parser.parseInline(token.dd)}</dd>`;
  },
  childTokens: ['dt', 'dd'],                    // Any child tokens to be visited by walkTokens
};

function walkTokens(token) {                        // Post-processing on the completed token tree
  if (token.type === 'strong') {
    token.text += ' walked';
    token.tokens = this.Lexer.lexInline(token.text)
  }
}
marked.use({ extensions: [descriptionList, description], walkTokens });

// EQUIVALENT TO:

marked.use({ extensions: [descriptionList] });
marked.use({ extensions: [description]     });
marked.use({ walkTokens })

console.log(marked.parse('A Description List:\n'
                + ':   Topic 1   :  Description 1\n'
                + ': **Topic 2** : *Description 2*'));
```

**Output**

```
<p>A Description List:</p>
<dl>
<dt>Topic 1</dt><dd>Description 1</dd>
<dt><strong>Topic 2 walked</strong></dt><dd><em>Description 2</em></dd>
</dl>
```

---

## Async Marked : `async`

Marked will return a promise if the `async` option is true. The `async` option will tell marked to await any `walkTokens` functions before parsing the tokens and returning an HTML string.

Simple Example:

```
const walkTokens = async (token) => {
  if (token.type === 'link') {
    try {
      await fetch(token.href);
    } catch (ex) {
      token.title = 'invalid';
    }
  }
};

marked.use({ walkTokens, async: true });

const markdown = `
[valid link](https://example.com)

[invalid link](https://invalidurl.com)
`;

const html = await marked.parse(markdown);
```

Custom Extension Example:

```
const importUrl = {
  extensions: [{
    name: 'importUrl',
    level: 'block',
    start(src) { return src.indexOf('\n:'); },
    tokenizer(src) {
      const rule = /^:(https?:\/\/.+?):/;
      const match = rule.exec(src);
      if (match) {
        return {
          type: 'importUrl',
          raw: match[0],
          url: match[1],
          html: '' // will be replaced in walkTokens
        };
      }
    },
    renderer(token) {
      return token.html;
    }
  }],
  async: true, // needed to tell marked to return a promise
  async walkTokens(token) {
    if (token.type === 'importUrl') {
      const res = await fetch(token.url);
      token.html = await res.text();
    }
  }
};

marked.use(importUrl);

const markdown = `
# example.com

:https://example.com:
```

```
`;

const html = await marked.parse(markdown);
```

## The Lexer

The lexer takes a markdown string and calls the tokenizer functions.

## The Parser

The parser takes tokens as input and calls the renderer functions.

## Access to Lexer and Parser

You also have direct access to the lexer and parser if you so desire.

```
const tokens = marked.lexer(markdown, options);
console.log(marked.parser(tokens, options));
```

```
const lexer = new marked.Lexer(options);
const tokens = lexer.lex(markdown);
console.log(tokens);
console.log(lexer.tokenizer.rules.block); // block level rules used
console.log(lexer.tokenizer.rules.inline); // inline level rules used
console.log(marked.Lexer.rules.block); // all block level rules
console.log(marked.Lexer.rules.inline); // all inline level rules
```

```
$ node
> require('marked').lexer('> I am using marked.')
[
  {
    type: "blockquote",
    raw: "> I am using marked.",
    tokens: [
      {
        type: "paragraph",
        raw: "I am using marked.",
        text: "I am using marked.",
        tokens: [
          {
            type: "text",
            raw: "I am using marked.",
            text: "I am using marked."
          }
        ]
      }
    ]
  },
  links: {}
]
```

The Lexer builds an array of tokens, which will be passed to the Parser. The Parser processes each token in the token array:

```
import { marked } from 'marked';

const md = `
  # heading

  [link][1]

  [1]: #heading "heading"
`;

const tokens = marked.lexer(md);
console.log(tokens);

const html = marked.parser(tokens);
console.log(html);
```

```
[
  {
    type: "heading",
    raw: "  # heading\n\n",
    depth: 1,
    text: "heading",
    tokens: [
      {
        type: "text",
        raw: "heading",
        text: "heading"
      }
    ]
  },
  {
    type: "paragraph",
    raw: "  [link][1]",
    text: "  [link][1]",
    tokens: [
      {
        type: "text",
        raw: "  ",
        text: "  "
      },
      {
        type: "link",
        raw: "[link][1]",
        text: "link",
        href: "#heading",
        title: "heading",
        tokens: [
          {
            type: "text",
            raw: "link",
            text: "link"
          }
        ]
      }
    ]
  },
  {
    type: "space",
    raw: "\n\n"
  },
  links: {
    "1": {
      href: "#heading",
      title: "heading"
    }
  }
]
<h1 id="heading">heading</h1>
```

```
<p>  <a href="#heading" title="heading">link</a></p>
```