micromark

main passing coverage 100% downloads 16M/month minzipped size 26.4 kB

sponsors 23 backers 33 chat discussions

The smallest CommonMark compliant markdown parser with positional info and concrete tokens.

- ✓ **compliant** (100% to CommonMark)
- ✓ **extensions** (GFM, directives, footnotes, frontmatter, math, MDX.js)
- ✓ **safe** (by default)
- ✓ **small** (smallest CM parser that exists)
- ✓ **robust** (1800+ tests, 100% coverage, fuzz testing)

Intro

micromark is a long awaited markdown parser. It uses a state machine to parse the entirety of markdown into concrete tokens. It's the smallest 100% CommonMark compliant markdown parser in JavaScript. It was made to replace the internals of `remark-parse`, the most popular markdown parser. Its API compiles to HTML, but its parts are made to be used separately, so as to generate syntax trees (`mdast-util-from-markdown`) or compile to other output formats. It's in open beta: up next are CMSM and CSTs.

- for updates, see Twitter
- for more about us, see `unifiedjs.com`

* for questions, see Discussions
* to help, see contribute or sponsor below

Contents

Install

npm:

```
npm install micromark
```

Use

Typical use (buffering):

```
var micromark = require('micromark')

console.log(micromark('## Hello, *world*!'))
```

Yields:

```
<h2>Hello, <em>world</em>!</h2>
```

The same can be done with ESM (in Node 10+, browsers that support it, or with a bundler), in an `example.mjs` file, like so:

```
import micromark from 'micromark'

console.log(micromark('## Hello, *world*!'))
```

You can pass extensions (in this case `micromark-extension-gfm`):

```
var micromark = require('micromark')
var gfmSyntax = require('micromark-extension-gfm')
var gfmHtml = require('micromark-extension-gfm/html')

var doc = '* [x] contact@example.com ~~strikethrough~~'

var result = micromark(doc, {
  extensions: [gfmSyntax()],
  htmlExtensions: [gfmHtml]
})

console.log(result)
```

Yields:

```
<ul>
<li><input checked="" disabled="" type="checkbox"> <a
href="mailto:contact@example.com">contact@example.com</a> <del>strikethrough</del>
</li>
</ul>
```

Streaming interface:

```
var fs = require('fs')
var micromarkStream = require('micromark/stream')

fs.createReadStream('example.md')
  .on('error', handleError)
  .pipe(micromarkStream())
  .pipe(process.stdout)

function handleError(err) {
  // Handle your error here!
  throw err
}
```

API

This section documents the API. The parts can be used separately, but this isn’t documented yet.

`micromark(doc[, encoding][, options])`

Compile markdown to HTML.

Parameters

`DOC`

Markdown to parse (`string` or `Buffer`)

`ENCODING`

[Character encoding](#) to understand `doc` as when it's a [`Buffer`](#) (`string` , default: `'utf8'`).

`OPTIONS.DEFAULTLINEENDING`

Value to use for line endings not in `doc` (`string` , default: first line ending or `'\n'`).

Generally, micromark copies line endings (`'\r'` , `'\n'` , `'\r\n'`) in the markdown document over to the compiled HTML. In some cases, such as `> a` , CommonMark requires that extra line endings are added: `<blockquote>\n<p>a</p>\n</blockquote>` .

`OPTIONS.ALLOWDANGEROUSHTML`

Whether to allow embedded HTML (`boolean` , default: `false`).

`OPTIONS.ALLOWDANGEROUSPROTOCOL`

Whether to allow potentially dangerous protocols in links and images (`boolean` , default: `false`). URLs relative to the current protocol are always allowed (such as, `image.jpg`). For links, the allowed protocols are `http` , `https` , `irc` , `ircs` , `mailto` , and `xmpp` . For images, the allowed protocols are `http` and `https` .

`OPTIONS.EXTENSIONS`

Array of syntax extensions ([`Array.<SyntaxExtension>`](#) , default: `[]`).

`OPTIONS.HTMLEXTENSIONS`

Array of HTML extensions ([`Array.<HtmlExtension>`](#) , default: `[]`).

Returns

`string` — Compiled HTML.

`micromarkStream(options?)`

Streaming interface of micromark. Compiles markdown to HTML. `options` are the same as the buffering API above. Available at `require('micromark/stream')` . Note that some of the work to parse markdown can be done streaming, but in the end buffering is required.

micromark does not handle errors for you, so you must handle errors on whatever streams you pipe into it. As markdown does not know errors, `micromark` itself does not emit errors.

Extensions

There are two types of extensions for micromark: [`SyntaxExtension`](#) and [`HtmlExtension`](#) . They can be passed in [`extensions`](#) or [`htmlExtensions`](#) , respectively.

`SyntaxExtension`

A syntax extension is an object whose fields are the names of hooks, referring to where constructs "hook" into. `content` (a block of, well, content: definitions and paragraphs), `document` (containers such as block quotes and lists), `flow` (block constructs such as ATX and setext headings, HTML, indented and fenced code, thematic breaks), `string` (things that work in a few places such as destinations, fenced code info, etc: character escapes and -

references), or `text` (rich inline text: autolinks, character escapes and -references, code, hard breaks, HTML, images, links, emphasis, strong).

The fields at such objects are character codes, mapping to constructs as values. The built in constructs are an extension. See it and the existing extensions for inspiration.

`HtmlExtension`

An HTML extension is an object whose fields are either `enter` or `exit` (reflecting whether a token is entered or exited). The values at such objects are names of tokens mapping to handlers. See the existing extensions for inspiration.

List of extensions

- `micromark/micromark-extension-directive` — support directives (generic extensions)
- `micromark/micromark-extension-footnote` — support footnotes
- `micromark/micromark-extension-frontmatter` — support frontmatter (YAML, TOML, etc)
- `micromark/micromark-extension-gfm` — support GFM (GitHub Flavored Markdown)
- `micromark/micromark-extension-gfm-autolink-literal` — support GFM autolink literals
- `micromark/micromark-extension-gfm-strikethrough` — support GFM strikethrough
- `micromark/micromark-extension-gfm-table` — support GFM tables
- `micromark/micromark-extension-gfm-tagfilter` — support GFM tagfilter
- `micromark/micromark-extension-gfm-task-list-item` — support GFM tasklists
- `micromark/micromark-extension-math` — support math
- `micromark/micromark-extension-mdx` — support MDX
- `micromark/micromark-extension-mdxjs` — support MDX.js
- `micromark/micromark-extension-mdx-expression` — support MDX (or MDX.js) expressions
- `micromark/micromark-extension-mdx-jsx` — support MDX (or MDX.js) JSX
- `micromark/micromark-extension-mdx-md` — support misc MDX changes
- `micromark/micromark-extension-mdxjs-esm` — support MDX.js import/exports

Syntax tree

A higher level project, `mdast-util-from-markdown`, can give you an AST.

```js
var fromMarkdown = require('mdast-util-from-markdown')

var result = fromMarkdown('## Hello, *world*!')

console.log(result.children[0])
```

Yields:

```js
{
  type: 'heading',
  depth: 2,
  children: [
    {type: 'text', value: 'Hello, ', position: [Object]},
    {type: 'emphasis', children: [Array], position: [Object]},
    {type: 'text', value: '!', position: [Object]}
  ],
  position: {
    start: {line: 1, column: 1, offset: 0},
    end: {line: 1, column: 19, offset: 18}
  }
}
```

Another level up is **remark**, which provides a nice interface and hundreds of plugins.

CommonMark

The first definition of “Markdown” gave several examples of how it worked, showing input Markdown and output HTML, and came with a reference implementation (`Markdown.pl`). When new implementations followed, they mostly followed the first definition, but deviated from the first implementation, and added extensions, thus making the format a family of formats.

Some years later, an attempt was made to standardize the differences between implementations, by specifying how several edge cases should be handled, through more input and output examples. This is known as CommonMark, and many implementations now work towards some degree of CommonMark compliancy. Still, CommonMark describes what the output in HTML should be given some input, which leaves many edge cases up for debate, and does not answer what should happen for other output formats.

micromark passes all tests from CommonMark and has many more tests to match the CommonMark reference parsers. Finally, it comes with CMSM, which describes how to parse markup, instead of documenting input and output examples.

Grammar

The syntax of markdown can be described in Backus–Naur form (BNF) as:

```
markdown = .*
```

No, that’s not a typo: markdown has no syntax errors; anything thrown at it renders *something*.

Test

micromark is tested with the ~650 CommonMark tests and more than 1.2k extra tests confirmed with CM reference parsers. These tests reach all branches in the code, thus this project has 100% coverage. Finally, we use fuzz testing to ensure micromark is stable, reliable, and secure.

To build, format, and test the codebase, use `$ npm test` after clone and install. The `$ npm run test-api` and `$ npm run test-coverage` scripts check the unit tests and their coverage, respectively. The `$ npm run test-types` script checks TypeScript definitions.

The `$ npm run test-fuzz` script does fuzz testing for 15 minutes. The timeout is provided by GNU coreutils **timeout(1)**, which might not be available on your system. Either install it or remove it from the script.

Size & debug

micromark is really small. A ton of time went into making sure it minifies well, by the way code is written but also through custom build scripts to pre-evaluate certain expressions. Furthermore, care went into making it compress well with GZip and Brotli.

Normally, you’ll use the pre-evaluated version of micromark, which is published in the `dist/` folder and has entries in the root. While developing or debugging, you can switch to use the source, which is published in the `lib/` folder, and comes instrumented with assertions and debug messages. To see debug messages, run your script with a `DEBUG` env variable, such as with `DEBUG="micromark" node script.js`.

To generate the codebase, use `$ npm run generate` after clone and install. The `$ npm run generate-dist` script specifically takes `lib/` and generates `dist/`. The `$ npm run generate-size` script checks the bundle size of `dist/`.

Comparison

There are many other markdown parsers out there, and maybe they’re better suited to your use case! Here is a short comparison of a couple of ’em in JavaScript. Note that this list is made by the folks who make `micromark` and `remark`, so there is some bias.

Note: these are, in fact, not really comparable: micromark (and remark) focus on completely different things than other markdown parsers do. Sure, you can generate HTML from markdown with them, but micromark (and remark) are created for (abstract or concrete) syntax trees—to inspect, transform, and generate content, so that you can make things like MDX, Prettier, or Gatsby.

MICROMARK

micromark can be used in two different ways. It can either be used, optionally with existing extensions, to get HTML pretty easily. Or, it can give tremendous power, such as access to all tokens with positional info, at the cost of being hard to get into. It’s super small, pretty fast, and has 100% CommonMark compliance. It has syntax extensions, such as supporting 100% GFM compliance (with `micromark-extension-gfm`), but they’re rather complex to write. It’s the newest parser on the block.

If you’re looking for fine grained control, use micromark.

REMARK

remark is the most popular markdown parser. It’s built on top of `micromark` and boasts syntax trees. For an analogy, it’s like if Babel, ESLint, and more, were one project. It supports the syntax extensions that micromark has (so it’s 100% CM compliant and can be 100% GFM compliant), but most of the work is done in plugins that transform or inspect the tree. Transforming the tree is relatively easy: it’s a JSON object that can be manipulated directly. remark is stable, widely used, and extremely powerful for handling complex data.

If you’re looking to inspect or transform lots of content, use remark.

MARKED

marked is the oldest markdown parser on the block. It’s been around for ages, is battle tested, small, popular, and has a bunch of extensions, but doesn’t match CommonMark or GFM, and is unsafe by default.

If you have markdown you trust and want to turn it into HTML without a fuss, use marked.

MARKDOWN-IT

markdown-it is a good, stable, and essentially CommonMark compliant markdown parser, with (optional) support for some GFM features as well. It’s used a lot as a direct dependency in packages, but is rather big. It shines at syntax extensions, where you want to support not just markdown, but *your* (company’s) version of markdown.

If you’re in Node and have CommonMark-compliant (or funky) markdown and want to turn it into HTML, use markdown-it.

OTHERS

There are lots of other markdown parsers! Some say they're small, or fast, or that they're CommonMark compliant — but that's not always true. This list is not supposed to be exhaustive. This list of markdown parsers is a snapshot in time of why (not) to use (alternatives to) `micromark`: they're all good choices, depending on what your goals are.

Version

The open beta of micromark starts at version `2.0.0` (there was a different package published on npm as `micromark` before). micromark will adhere to semver at `3.0.0`. Use tilde ranges for now: `"micromark": "~2.10.1"`.

Security

The typical security aspect discussed for markdown is cross-site scripting (XSS) attacks. It's safe to compile markdown to HTML if it does not include embedded HTML nor uses dangerous protocols in links (such as `javascript:` or `data:`). micromark is safe by default when embedded HTML or dangerous protocols are used too, as it encodes or drops them. Turning on the `allowDangerousHtml` or `allowDangerousProtocol` options for user-provided markdown opens you up to XSS attacks.

Another aspect is DDoS attacks. For example, an attacker could throw a 100mb file at micromark, in which case the JavaScript engine will run out of memory and crash. It is also possible to crash micromark with smaller payloads, notably when thousands of links, images, emphasis, or strong are opened but not closed. It is wise to cap the accepted size of input (500kb can hold a big book) and to process content in a different thread or worker so that it can be stopped when needed.

Using extensions might also be unsafe, refer to their documentation for more information.

For more information on markdown sanitation, see `improper-markup-sanitization.md` by **@chalker**.

See `security.md` in `micromark/.github` for how to submit a security report.

Contribute

See `contributing.md` in `micromark/.github` for ways to get started. See `support.md` for ways to get help.

This project has a code of conduct. By interacting with this repository, organisation, or community you agree to abide by its terms.

Sponsor

Support this effort and give back by sponsoring on OpenCollective!

Salesforce 🏅

salesforce						
Gatsby 🥇	Vercel 🥇	Netlify	Holloway	ThemeIsle	Boost Hub	Expo
You?						

Origin story

Over the summer of 2018, micromark was planned, and the idea shared in August with a couple of friends and potential sponsors. The problem I (**@wooorm**) had was that issues were piling up in remark and other repos, but my day job (teaching) was fun, fulfilling, and deserved time too. It was getting hard to combine the two. The thought was to feed two birds with one scone: fix the issues in remark with a new markdown parser (codename marydown) while being financially supported by sponsors building fancy stuff on top, such as Gatsby, Contentful, and Vercel (ZEIT at the time). **@johno** was making MDX on top of remark at the time (important historical note: several other folks were working on JSX + markdown too). We bundled our strengths: MDX was getting some traction and we thought together we could perhaps make something sustainable.

In November 2018, we launched with the idea for micromark to solve all existing bugs, sustaining the existing hundreds of projects, and furthering the exciting high-level project MDX. We pushed a single name: unified (which back then was a small but essential part of the chain). Gatsby and Vercel were immediate sponsors. We didn't know whether it would work, and it worked. But now you have a new problem: you are getting some financial support (much more than other open source projects) but it's not enough money for rent, and too much money to print stickers with. You still have your job and issues are still piling up.

At the start of summer 2019, after a couple months of saving up donations, I quit my job and worked on unified through fall. That got the number of open issues down significantly and set up a strong governance and maintenance system for the collective. But when the time came to work on micromark, the money was gone again, so I contracted through winter 2019, and in spring 2020 I could do about half open source, half contracting. One of the contracting gigs was to write a new MDX parser, for which I also documented how to do that with a state machine in prose. That gave me the insight into how the same could be done for markdown: I drafted CMSM, which was some of the core ideas for micromark, but in prose.

In May 2020, Salesforce reached out: they saw the bugs in remark, how micromark could help, and the initial work on CMSM. And they had thousands of Markdown files. In a for open source uncharacteristic move, they decided to fund my work on micromark. A large part of what maintaining open source means, is putting out fires, triaging issues, and making sure users and sponsors are happy, so it was amazing to get several months to just focus and make something new. I remember feeling that this project would probably be the hardest thing I'd work on: yeah, parsers are pretty difficult, but markdown is on another level. Markdown is such a giant stack of edge cases on edge cases on even more weirdness, what a mess. On August 20, 2020, I released 2.0.0, the first working version of micromark. And it's hard to describe how that moment felt. It was great.

License