

Developer's guide to Markdown extensions



- [The Basics](#)
 - [How to structure an extension?](#)
 - [The setup protocol](#)
 - [Putting it all together](#)
 - [Discourse specific extensions](#)
 - [Generating hashtag references in Ruby](#)
 - [BBCode](#)
 - [Inline BBCode rules](#)
 - [Block BBCode rules](#)
 - [Handling text replacements](#)
-

Discourse uses a Markdown engine called [Markdown-it](#).

Here are some dev notes that will help you either fix bugs in core or create your new plugins.

The Basics

Discourse only contains a few helpers on top of the engine, so the vast majority of learning that needs to be done, is understanding Markdown It.

The [docs directory](#) contains the current documentation.

I strongly recommend reading:

- The [architecture document](#) to understand at a top level how the engine works.
- [Development](#) for basic development guidelines
- [API documentation](#) for a very detailed reference

- And finally, the [source code](#) which is very well documented and clear.

While I develop extensions for the engine I usually open up a second editor looking at existing rules. The engine consists of a long list of rules and each rule is in a dedicated file that is reasonably easy to follow.

If I am working on an inline rule I will think of what existing inline rule works more or less like it and base my work on it.

Keep in mind, you can sometimes get away with just changing a renderer to get desired functionality which is usually much easier.

How to structure an extension?

When the markdown engine initializes it searches through all the modules.

If any module is called `/discourse-markdown\|markdown-it\|` (meaning it lives in a discourse-markdown or markdown-it directory) it will be a candidate for initialization.

If the module **exports** a method called `setup` it will be called by the engine during initialization.

The setup protocol

```
/my-plugins/assets/javascripts/discourse-markdown/awesome-extension.js
```

```
export function setup(helper) {  
  // ... your code goes here  
}
```

A `setup` method gets access to a helper object it can use for initialization. This contains the following methods and vars:

- `bool markdownIt` : this property is set to `true` when the new engine is in use. For proper backwards compatibility you want to check it.
- `registerOptions(cb(opts, siteSettings, state))` : the provided function is called before the markdown engine is initialized, you can use it to determine if to enable or disable the engine.
- `allowList([spec, ...])` : this method is used to allowlist HTML with our sanitizer.
- `registerPlugin(func(md))` : this method is used to register a [Markdown It plugin](#).

Putting it all together

```
function amazingMarkdownItInline(state, silent) {
  // standard markdown it inline extension goes here.
  return false;
}

export function setup(helper) {
  if(!helper.markdownIt) { return; }

  helper.registerOptions((opts,siteSettings)=>{
    opts.features['my_extension'] = !!siteSettings.my_extension_enabled;
  });

  helper.allowList(['span.amazing', 'div.amazing']);

  helper.registerPlugin(md=>{
    md.inline.push('amazing', amazingMarkdownItInline);
  });
}
```

Discourse specific extensions

Generating hashtag references in Ruby

Use `HashtagAutocompleteService#hashtags_for` to build references for generated Markdown:

```
HashtagAutocompleteService.new(user.guardian).hashtags_for("tag", ["bug", "support"]).join("
")
```

If a visible category has the slug `bug`, the result is `#bug::tag #support`. Pass names already filtered for the viewer's visibility. Order and case are preserved, collisions are checked against higher-priority types, and a name the hashtag rule cannot match falls back to plain text.

BBCode

Discourse contains 2 rulers you can use for custom BBCode tags. An inline and block level ruler.

Inline bbcode rules are ones that live in an inline paragraph like `[b]bold[/b]`

Block level rules apply to multiple lines of text like:

```
[poll]
- option 1

- options 2
[/poll]
```

`md.inline.bbcode.ruler` holds a list of inline rules that are applied in order.

`md.block.bbcode.ruler` holds a list of block level rules

There are many examples for inline rules at: [bbcode-inline.js](#)

[Quotes](#) and polls are good examples of bbcode block rules.

Inline BBCode rules

Inline BBCode rules are an object containing information about how to handle a tag.

For example:

```
md.inline.bbcode.ruler.push("underline", {
  tag: "u",
  wrap: "span.bbcode-u",
});
```

Will cause

```
test [u]test[/u]
```

To be converted to:

```
test <span class="bbcode-u">test</span>
```

Inline rules can either wrap or replace text. When wrapping you can also pass in a function to gain extra flexibility.

```
md.inline.bbcode.ruler.push("url", {
  tag: "url",
  wrap: function (startToken, endToken, tagInfo, content) {
    const url = (tagInfo.attrs["_default"] || content).trim();
```

```

if (simpleUrlRegex.test(url)) {
    startToken.type = "link_open";
    startToken.tag = "a";
    startToken.attrs = [
        ["href", url],
        ["data-bbcode", "true"],
    ];
    startToken.content = "";
    startToken.nesting = 1;

    endToken.type = "link_close";
    endToken.tag = "a";
    endToken.content = "";
    endToken.nesting = -1;
} else {
    // just strip the bbcode tag
    endToken.content = "";
    startToken.content = "";

    // edge case, we don't want this detected as a onebox if auto linked
    // this ensures it is not stripped
    startToken.type = "html_inline";
}

return false;
},
});

```

The wrapping function provides access to:

- The tagInfo, which is a dictionary of key/values specified via bbcode.

`[test=testing]` -> `{_default: "testing"}` `[test a=1]` -> `{a: "1"}`

- The token starting the inline
- The token finishing the inline
- The content of the bbcode inline

Using this information you can handle all sort of wrapping needs.

Occasionally you may want to replace the entire BBCode block, for that you can use `replace`

```

md.inline.bbcode.ruler.push("code", {
    tag: "code",

```

```
replace: function (state, tagInfo, content) {
  let token;
  token = state.push("code_inline", "code", 0);
  token.content = content;
  return true;
},
});
```

In this case we are replacing an entire `[code]code block[code]` with a single `code_inline` token.

Block BBCode rules

Block bbcode rules allow you to replace an entire block. The block APIs are the same for simple cases:

```
md.block.bbcode.ruler.push("happy", {
  tag: "happy",
  wrap: "div.happy",
});
```

```
[happy]
hello
[/happy]
```

will become

```
<div class="happy">hello</div>
```

The function wrapper has a slightly different API cause there are not wrapping tokens.

```
md.block.bbcode.ruler.push("money", {
  tag: "money",
  wrap: function (token, tagInfo) {
    token.attrs = [{"data-money", tagInfo.attrs["_default"]}];
    return true;
  },
});
```

```
[money=100]
**test**
[/money]
```

Will become

```
<div data-money="100">
  <b>test</b>
</div>
```

You can gain full control over block rendering with `before` and `after` rule, this allows you to do stuff like double nest a tag and so on.

```
md.block.bbcode.ruler.push("ddiv", {
  tag: "ddiv",
  before: function (state, tagInfo) {
    state.push("div_open", "div", 1);
    state.push("div_open", "div", 1);
  },
  after: function (state) {
    state.push("div_close", "div", -1);
    state.push("div_close", "div", -1);
  },
});
```

```
[ddiv]
test
[/ddiv]
```

will become

```
<div>
  <div>test</div>
</div>
```

Handling text replacements

Discourse ships with an extra special core rule for applying regular expressions to text.

```
md.core.textPostProcess.ruler
```

To use:

```
md.core.textPostProcess.ruler.push("onlyfastcars", {
  matcher: /(car)|(bus)/, //regex flags are NOT supported
```

```
onMatch: function (buffer, matches, state) {  
  let token = new state.Token("text", "", 0);  
  token.content = "fast " + matches[0];  
  buffer.push(token);  
},  
});
```

I like cars and buses

Will become

<p>I like fast cars and fast buses</p>

#2

2026-10-31 13:28:01 UTC

2026-10-31 13:28:02 UTC