Writing Markdown extensions (1): Customizing renderer behavior

Markdown gives you two options to work with specialized content: You can write it as HTML directly or you can use plugins for your Markdown tools that will produce the same result from some specific combination of text symbols.

Until now I've always used HTML for things like figures and video embeds from YouTube or Codepen. It's easier, I've created snippets in VSCode to handle them and my muscle memory already knows the shortcuts and how to write the code by hand.

For security reasons this may not always be a good idea, specially if you accept third-party contributions. There's nothing stoping a malicious author from inserting scripts into the HTML that would run when a reader views the page.

I've chosen to experiment with Markdown-it as a standalone editor and see how easy it is to write plugins for them (or not, see Markdown to HTML standalone converter tool for my reservations about the project leadership).

Modifying existing renderer rules

The easiest way to create Markdown-it plugins is to customize the way that the Markdown renderer works. The following example will add `loading="lazy"` to all images so they will lazy load in Chromium-based browsers.

```
const imageDefaultRender =
  md.renderer.rules.image ||
  function (tokens, idx, options, env, self) {
    return self.renderToken(tokens, idx, options);
  };

md.renderer.rules.image = function (tokens, idx, options, env, self) {
  const loadingIndex = tokens[idx].attrIndex('loading');
```

```
  if (loadingIndex < 0) {
    tokens[idx].attrPush(['loading', 'lazy']);
  } else {
    tokens[idx].attrs[loadingIndex][1] = 'lazy';
  }

  return defaultRender(tokens, idx, options, env, self);
};
```

This is an all-or-nothing deal. Either all the images get it or you have to manually edit the resulting HTML code. For native lazy loading this is not bad since browsers will not lazy load the images that are on the viewport when the page initially loads but it may be an issue for other applications.

It also present the issue that, if you change one attribute, any other attribute will have its value reset. This happened with the `alt` attribute. To fix it we had to add a second if/then/else block to handle the `alt` attribute. I'm not 100% sure if you'd have to do the same for other existing attributes.

```
const imageDefaultRender =
  md.renderer.rules.image ||
  function (tokens, idx, options, env, self) {
    return self.renderToken(tokens, idx, options);
  };

md.renderer.rules.image = function (tokens, idx, options, env, self) {
  const loadingIndex = tokens[idx].attrIndex('loading');
  const altIndex = tokens[idx].attrIndex('alt');

  if (loadingIndex < 0) {
    tokens[idx].attrPush(['loading', 'lazy']);
  } else {
    tokens[idx].attrs[loadingIndex][1] = 'lazy';
  }

  'alt'// We test for presence and length
  // to prevent a crash if the
```

```
  // attribute is empty (lenght = 0) (altIndex < 0 && altIndex.length > 0
    tokens[idx].attrPush(['alt', tokens[0].children[0].content]);
  } else {
    tokens[idx].attrs[altIndex][1] = tokens[0].children[0].content;
  }

  return imageDefaultRender(tokens, idx, options, env, self);
};
'alt'
```

Another example is to add `noopener`, `nofollow` and `noreferrer` attributes to external links.

```
var defaultRender =
  md.renderer.rules.link_open ||
  function (tokens, idx, options, env, self) {
    return self.renderToken(tokens, idx, options);
  };

md.renderer.rules.link_open = function (tokens, idx, options, env, self) {
  var aIndex = tokens[idx].attrIndex('rel');

  if (aIndex < 0) {
    tokens[idx].attrPush(['rel', 'noopener noreferrer nofollow']);
  } else {
    tokens[idx].attrs[aIndex][1] = 'noopener noreferrer nofollow';
  }

  return defaultRender(tokens, idx, options, env, self);
};
```

Again, this is an all or nothing proposition but this time, it's not harmless. We don't want to add the attributes to internal links so we have to code around the problem.

> First we create two functions that will define if a link is internal or not.
>
> The first one gets the domain for the site (the domain is defined as everything

between the `//` and the first `/` in the URL, assuming that external links will look like this `http://example.com/demo.html`)

The second functions defines an internal link as one without a domain.

```
function getDomain(href) {
  let domain = href.split('//')[1];
  if (domain) {
    domain = domain.split('/')[0].toLowerCase();
    return domain || null;
  }
  return'/'ll;
}

function isInternalLink(href) {
  let domain = getDomain(href);
  return domain === null;
}
```

We then modify our link_open renderer so it will only add the attributes to links that are **not** internal.

```
// Remember old renderer, if overridden, or proxy to default renderer
var defaultRender =
  md.renderer.rules.link_open ||
  function (tokens, idx, options, env, self) {
    return self.renderToken(tokens, idx, options);
  };

md.renderer.rules.link_open = function (tokens, idx, options, env, self) {
  var linkIndex = tokens[idx].attrIndex('rel');
  var href = tokens[idx].attrGet('href');

  'rel'// Only do it if the link is not internal (!isInternalLink(href)) {
    if (linkIndex < 0) {
      tokens[idx].attrPush(['rel', 'noopener noreferrer nofollow']);
    } else {
      tokens[idx].attrs[linkIndex][1] = 'noopener noreferrer nofollow';
```

```
    }
  }
  // pa'rel'// pass token to default renderer.
  return defaultRender(tokens, idx, options, env, self);
};
```

Further improvements

Right now our code is very rigid. It adds the attributes we tell it to and requires further work to add the customizations. Based on my limited knowledge of how the parser internals work I don't know if it's possible to further customize the rules customizations with customizations that fall outside the Markdown parser or if this would fall under creating new Markdown elements

Writing Markdown extensions (2): Thoughts and ideas (so far)

Before jumping into writing our own Markdown elements and all the complexity that goes with them, let's stop and see where are are.

Markdown rule customization gives you a lot of flexibility regarding customizing built-in rules and elements. It's an all or nothing proposition, either all elements have the attributes you're customizing or you code around to make sure only some do (like the link example where we only add attribute to external links).

Most of the time, these customizations will be enough, but not always. There may be times when we want an element that is not available on standard Markdown or we don't want to modify the default element to suit our needs because we still need it in its default form.

In that case we'd have to write our own custom Markdown element. This is much harder than the code we've written so far because it digs deeper into Markdown-it internals and because the documentation on how to do it is scarce and almost non-existent, leaving beginners to try and figure out things from the source code.

Looking at existing plugins or the Markdown-It engine doesn't help as much as I thought it would. Markdown-it itself is well documented but it assumes you know everything that's going on and that's not always the case.

As I get more comfortable with the code I'll write another post covering the details of the process.

When to write your own and when to trust third parties

It is likely that if you need to do something other people will want to do it too, so it begs the question: When do you write your own code and when do you use someone else's?

A related problem with Markdown-it plugins is that many of the plugins available on NPM are copies of existing packages, some scoped to specific users (instead of `package` it becomes `@user/package`) and some just forked from the original for no reason that's apparent to me. This makes it harder to figure out which package use on your projects, making it even more tempting to write your own.

We'll cover specifics of how to write a custom plugin in a later post but as mentioned earlier, it's not an easy task because there is little to no documentation on how to do it. You're expected to understand the code well enough to make it happen.

Writing Markdown extensions (3): Writing our own Markdown elements

The other way to create custom Markdown and the corresponding HTML is to create completely custom Markdown elements.

This is more complicated since we now have to parse the input, not just match an existing renderer but with the complexity comes a much more powerful toolset.

In pseudocode, as far as I understand it, the process goes something like this:

1. Design the new element and the resulting markup
    1. Decide what characters will make up the open and closing strings for the element
    2. Decide what HTML will the rule produce
2. Write custom rules to process the new markup we want to use
3. Add the custom rules to the parser pipeline so they'll be used when we render documents to HTML
4. Run the renderer to generate HTML