# Customizing Markdown-it

In the last post we modified a Gulp workflow to include Markdown-it directly. But there are parts that can't be done with plugins and are not part of the Markdown-It API.

This post will explore some of the customizations I've done with the Markdown-It API.

## Manipulating images: adding the loading attribute

The first example is a simple one. We will add the `loading="lazy"` attribute to all images so that browsers that support native lazy loading will use the feature and only load the image when it's about to appear in the viewport or already in the viewport.

```
md.renderer.rules.image = function (tokens, idx, options, env, slf) {
  const token = tokens[idx];
  token.attrs[token.attrIndex('alt')][1] = slf.renderInlineAsText(token.ch
  token.attrSet('loading', 'lazy');
  return slf.renderToken(tokens, idx, options)
}
```

Just by adding this attribute we've made lazy loading images the default behavior. Other than images that appear in above the fold when the page loads, all other images will load only when they appear in the viewport.

## Adding attributes to links

Add `rel="noopener norefeerrer"` attributes to links for SEO purposes

The final format should be `<a href="https://example.com/" rel="noopener noreferrer">Example Link</a>`.

Using the `noopener` tells the browser to open a link in a new tab without

providing backdoor access to the webpage that opened the link. This is achieved by not setting the `window.opener` property thus returning a null value.

`noreferrer` will hide referrer information when the link is clicked. For example, if links to your post from their webpage and use the noreferrer, and then users click on that link, you will not be able to tell where did those users come from. In your analytics software (say, Google Analytics), this will appear as direct traffic, not as a referral.

There is an additional attribute that I chose not to use.

In general, when one page links to yours, it adds credibility to your site and signals to search engines that they value your website. If a high authority webpage links to you it is endorsing you, and Google/Bing will consider the endorsement a ranking factor. Google uses the term PageRank as a measure of quantity and quality of links.

Adding `nofollow` signals search engines that you don't want to pass your 'endorsement' to the page you are linking to.

See Explained: noopener, noreferrer, and nofollow Values for more information.

The code joins the existing attributes for the link with the new rel="noopener noreferrer" attribute.

```
defaultLinkOpenRenderer = md.renderer.rules.link_open || proxy;

md.renderer.rules.link_open = function(tokens, idx, options, env, self) {
  tokens[idx].attrJoin("rel", "noopener noreferrer");
  return defaultLinkOpenRenderer(tokens, idx, options, env, self)
};
```

# Adding classes to list and list items

There are a few things that I like doing with lists. Instead of trying to add attribute after attribute to control styles, I pass one or more classes to either the list itself of individual list items.

The first example adds a class attribute to the ordered (bulleted list). This could be used to change the numbering schema used on the list or style the list itself.

```
const proxy = (tokens, idx, options, env, self) => self.renderToken(tokens
const defaultOrderedListOpenRenderer = md.renderer.rules.ordered_list_oper

md.renderer.rules.ordered_list_open = function(tokens, idx, options, env,
   tokens[idx].attrJoin("class", "sample-class");
   return defaultOrderedListOpenRenderer(tokens, idx, options, env, self)
};
```

The second example styles list items. This is an all or nothing proposition. Either all list items get the class or none of them. Since the styles for bulleted and ordered list are differents the class shouldn't have an effect where we don't want it to, it just adds unnecessary bytes to the page

```
const proxy = (tokens, idx, options, env, self) => self.renderToken(tokens

const defaultListItemOpenRenderer = md.renderer.rules.list_item_open || pr

md.renderer.rules.list_item_open = function(tokens, idx, options, env, se
  tokens[idx].attrJoin("class", "decimal-zero-padded");

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

We're using lists as an example. We could use a similar technique with other elements like tables, and paragraphs.

# Wrapping images in figure elements

The hardest one of these extensions is wrapping images in figure elements.

The first pass is easy enough, although it is also misleading since it doesn't do exactly what we want.

The first version of this task, wraps the image in a `figure` element as part of

the return statement.

```
const md = require('markdown-it')();

md.renderer.rules.image = function (tokens, idx, options, env, slf) {
  const token = tokens[idx]
  token.attrs[token.attrIndex('alt')][1] = slf.renderInlineAsT'alt'oken.ch
  token.attrSet('loading', 'lazy')

  return 'loading'`<figure>
  ${slf.renderToken(tokens, idx, options)}
  <figcaption>${token.attrs[token.attrIndex('a'alt'[1]}</figcaption>
</figure>`
}
```

The surprise is that Markdown (both the original documents and the current Commonmark specification) consider images to be inline elements, rather than elements that can be either block or inline (like smileys or image emojis).

As a result, the parser wraps the image in a p element.

So the next, and significantly harder task, is to remove the paragraphs and replace them with a `figure` element, which is sematically more appropriate than a paragraph.

At the same time, it would be nice if we could add a cpation to the image using the `figcaption` element.

# Wrapping images in figure elements (Version 2)

The second version of this code is a bit more complicated.

In addition to the lazy loading attribute, it takes the value of the `alt` attribute and uses it as the caption for the image

```
md.renderer.rules.image = function (tokens, idx, options, env, slf) {
```

```
  const token = tokens[idx]
  token.attrs[token.attrIndex('alt')][1] = slf.renderInlineAsText(token.ch
  token.attrSet('loading', 'lazy')

  return 'loading'`<figure>
  ${slf.renderToken(tokens, idx, options)}
  <figcaption>${token.attrs[token.attrIndex('a'alt'[1]}</figcaption>
</figure>`
}
```

It will produce the following HTML code:

```
<p><figure>
  <img src="demo.png"
    alt="demo"
    loading="lazy">
  <figcaption>demo</figcaption>
</figure></p>
```

It still doesn't get rid of the paragraphs, still working on figuring out how to do the replacement but, as it is, it works well enough without requiring a plugin.

# Wrapping images in figure elements using a third party plugin (take 1)

I haven't been able to find a way to remove the paragraph container around the image. When I was about ready to give up I came across a pointer to the markdown-it-custom-block plugin that lets me do exactly what I want and then some.

The original code using the plugin looks like this

```
const cb = require('markdown-it-custom-block');

md.use(cb, {
```

```
  img(raw) {
    const [index, alt, width, url] = raw.split('#');
    return '#'`<figure id="fig${index}">
  <img width='${width}' src="${url}" alt="${alt}">
  <figcaption>Fig ${index}: ${alt}</figcaption>
</figure>`;
  },
});
```

The Markdown for the image is more complicated. It is a list of four attributes separated by # characters representing:

- The index (position of the image on the page)
- The alt text displayed when the image is not loaded and also used as the figure caption
- The width of the image
- The full path to the image file

```
@[img](1#Sample alt content will also be used for captions#400px#./images/
```

This is more complicated than it needs to be, but it works.

# Wrapping images in figure elements using a third party plugin (take 2)

One of the first things I thought about was removing the index and letting CSS take care of the indexing for us. Hardcoding the index in Markdown is a bad idea. If we insert images then all the indexes change and we need to edit every single image to compensate.

Using CSS generated content we can automate the indexing and generation of the `Figure <id>:` content in the captions:

```
article {
  counter-reset: figures;
}
```

```
figure {
  counter-increment: figures;
}

figure figcaption:before {
  content: "Fig. " counter(figures) ": ";
}
"Fig. "
```

And it does what I set out to do, eliminate the paragraphs around images and use figures instead.

So now the final version of our image code is:

```
const cb = require('markdown-it-custom-block');

md.use(cb, {
  img(raw) {
    const [
      alt,
      width,
      url] = raw.split('#');
    return '#'`<figure>
  <img width='${width}' src="${url}" alt="${alt}">
  <figcaption>${alt}</figcaption>
</figure>`;
  },
});
```

And the required Markdown changes accordingly:

```
@[img](Sample alt content will also be used for captions#400px#./images/de
```