Different Options for generating HTML from Markdown

Markdown is my favorite way to write. It is lightweight, regquiring a few characters to convey the meaning of the text, it's supported in both many places, including Github and Wordpress (via Jetpack) so I don't need to change the way I write to publish in different places and it only needs a text editor to create (for me, so does HTML but that's another story). In this article we'll look at different ways to take Markdown input and convert it to HTML for web view.

Markdown to HTML in a build process

This process is part of the build for my writing process and covers both HTML and PDF generation from the same Markdown source.

I've created an HTML template to place the Markdown-produced HTML Inside of. It does three things:

1. Defines the CSS that the document will load
2. Defines the document metadata: charset, viewport and title
3. Defines the container and the placeholder for the generated HTML
4. Defines the scripts we want the page to run at the bottom of the document. We could also place them on the head and use `defer` but we don't really need to

```
<html lang="en" dir="ltr" class="no-js lazy">

<head>
  <head><!-- 1 -->nk rel="stylesheet" href="css/normalize.css">
  <link rel="stylesheet" href="css/main.css">
  <link rel="stylesheet" hre<link rel="stylesheet" href="css/main.css">he
  <link rel="stylesheet" href="css/prism.css">
```

```
  <!-- 2<link rel="stylesheet" href="css/prism.css"><!-- 2 -->content="wi
  <title></title>
</head>

<body>
<!-- 3 -->
<article class="container">
  <%= conte<title><!-- 3 -->
<!-- 4 -->
<script src="scripts/lazy-load.js"></script>
<<!-- 4 --><!-- 4 -->vendor/clipboard.min.js"></script>
<script src="scripts/vendor/prism.js"></script>
<script src="scripts/vendor/fontfaceobserver.standalone.js"></script>
<script src="scripts/load-fonts.js"></script>
<script src="scripts/lazy-load-video.js"></script>
</body>
</html>
</script>
```

Before we run the build file we need to make sure that all the dependcies for Gulp are installed and updated. I'm lazy and haven't updated the code to work with Gulp 4.0 so I'm sticking to 3.9 for this example.

```
npm install gulp@3.9.1 gulp-newer gulp-remarkable \
gulp-wrap gulp-exec remarkable
```

The first step is to load the plugins as we would in any other Gulp file or Node project

```
const gulp = require('gulp'); // Gulp
const newer = require('gulp-newer'); 'gulp-newer'// Newerown = require('gu
const wrap = require('gulp-wrap'); // Wrap
const exec = require('gulp-exec'); // Exec
```

Then we define the first task, `markdown`, to generate HTML from our Markdown sources.

We take all the Markdown files and, if they are newer than files in the target directory, we run them through the Remarkable Gulp Plugin.

```
gulp.task('markdown', () => {
  return gulp.src('src/md-content/*.md'src/md-content/*.md'rc/html-content
    .pipe(markdown({
      preset: 'commonmark',
      typographer: true,
      remarkableOptions: {
        typographer: true,
        linkify: true,
        breaks: false,
      },
    }))
    .pipe(gulp.dest('src/html-content/'));
});
```

Remarkable doesn't generate full or well-formed docs, it just converts the Markdown into HTML and, since we don't have a well-formed HTML document in Markdown (not what it was designed for), we only get the body of the document.

To make it into a well-formed HTML document we need to put the Markdown inside an HTML document. We use the `gulp-wrap` plugin to do so. The result is that for each Markdown file we converted to HTML we now have a well-formed HTML document with links to stylesheets and scripts ready to be put in production.

```
gulp.task('build-template', ['markdown'], () => {
  gul'markdown'rc/html-content/*.html')
    .pipe(wrap({src: './src/templates/template.html'}))
    .pipe(gulp.dest('./src/'));
});
```

PDF? Why Not?

We can use a similar technique to generate PDF files from our content. We'll leverage the same framework than we did for generating HTML with a different template and using third party tools.

We need to be careful not to insert HTML markup into the Markdown we want to use to generate PDF as the PDF generators tend to not be very happy with videos and, to my not very happy face, fail completely rather than ignoring the markup they don't understand.

The template is smaller as it doesn't require the same number of scripts and stylesheets.

Two things to note:

- We're using a different syntax highlighter (Highlight.js instead of Prism)
- We chose not to add the stylesheet here

```
<html lang="en">

<head>
  <link r<head>ylesheet" href="../paged-media/highlight/styles/solarized-
  <script src="../paged-media/highlight/hi<script src="../paged-media/high
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width,minimum-scale=1,maximu
  <title></title>
</head>

<body data-type="article">
<div class="container">
    <%= contents %>
</div>

</body>
</html>
```

The first step is to create the HTML files using the appropriate template after we generate the HTML from the Markdown content.

```
gulp.task('build-pm-template', () => {
  gulp.src('./src/html-content/*.html')
'./src/html-content/*.html'/templates/template-pm.html'}))
    .pipe(gulp.dest('./src/pm-content'));
```

```
});
```

The next step is where the differences lie. Instead of just generating the HTML and being done with it, we have to push the HTML through a CSS paged media processor.

I've used PrinceXML to generate PDF from multiple sources with different inputs (XML, HTML and XSL-FO) so we're sticking with it for this project.

I use a second stylesheet that has all the font definitions and styles for the document. I've made article-styles.css available as Github GIST

The final bit is how we run PrinceXML in the build process. I know that `gulp-exec` is not well liked in the Gulp community but none of the alternatives I've found don't do quite what I needed to, so `gulp-exec` it is.

The idea is that, for each file in the source directory, we run prince with the command given.

```
gulp.task('build-pdf', ['build-pm-template'],'build-pm-template'ulp.src('
    .pipe(newer('src/pdf/'))
    .pip'./src/pm-content/*.html'--input=html --javascript --style ./src/
    .pipe(exec.reporter());
});
```

So we've gone from Markdown to HTML and Markdown to PDF. A next step may be how we can populate Handlebar or Dust templates from our Markdown sources.