Markdown to HTML standalone tool

Ever since I started to work with Markdown as my authoring language, I've used Gulp as a conversion tool, when needed. The problem is that gulp-remarkable is poorly documented and has poor support for plugins because of it... I have yet to figure out how to implement the plugins.

Using Markdown-it as my Markdown parser and gulp-markdownit seems to be more forgiving and easier to use, still no Markdown-it plugins but the standalone tool works fine (as we'll describe later).

It is not perfect... the leadership of the project leaves a lot to be desired (reason why I'm not contributing to the project).

This Github Issue worries me. While it's true that the person filling the issue could have addressed it better but pointing people to source code is seldom the solution when you're asking for documentation. It leads to frustration and to people not wanting to use the code.

Documentation seems to be a chronic problem with open source projects but the response is still not "cool".

I also find this statement worrisome:

> markdown-it is the result of the decision of the authors who contributed to 99% of the Remarkable code to move to a project with the same authorship but new leadership (Vitaly and Alex). It's not a fork.

But that's exactly what a fork is. You stop working on the original, take the code and start doing your own work that might or might not remain compatible with the original. The point about autorship is debatable but that's a subject for another post.

According to Wikipedia:

> In software engineering, a project fork happens when

> developers take a copy of source code from one software package and start independent development on it, creating a distinct and separate piece of software.

Goals and ideas

Moving to a straight Node converter seems to be a daunting challenge where some parts are easy and some parts seem to be really complex. The steps that we want to duplicate are:

1. Convert Markdown files to HTML fragments and save the fragments to a different file
2. Insert the HTML fragments into a template file suitable for publishing to the web
3. (Optional) Insert the HTML fragments into a template file suitable for processing with PrinceXML or Vivliostyle
4. (Optional) convert the files generated in step 3 to PDF using PrinceXML
    1. Evaluate Vivliostyle as an open-source alternative

The Markdown to HTML converter

I've broken up the implementation in chunks to make it easier to write about it. This code addresses items one and two of our requirements list.

First we require all the plugins we'll need.

The code uses the `fs` and `process` built in Node modules so we register those first.

Next we register the Markdown-it plugins that we want to use for the project. Note that requiring the plugins doesn't mean Markdwon-it will use them

```
// Builtin modules
const proc = require('process');
const 'process'ire('fs');

'fs'// Markdown-it pluginsst abbr = require("markdown-it-abbr");
const anc = require("markdown-it-anchor");
```

```
const attrs = require("markdown-it-attrs""markdown-it-abbr"equire("markdo
const fn = require("markdown-it-footnote");
const figs = require("markdown-it-implicit-figures");
const kbd = require("markdown-it-kbd");
const prism = require("markdown-it-prism");
const toc = require("markdown-it-tab"markdown-it-block-embed"ist = requir

// M"markdown-it-task-lists"// Markdown
const md = require('markdown-it')('commonmark')
  .use(abbr)
  .use(anc, {
    permalink: true
  })
  .use(attrs)
  .use(embed, {
    containerClassName: 'video',
  })
  .use(fn)
  .use(figs, {
    dataType: false,
    figcaption: true,
    tabindex: false,
    link: false
  })
  .use(kbd)
  .use(prism)
  .use(toc, {
    includeLevel: [1,2,3]
  })
  .use(list)
```

I'm not 100% certain these are the plugins the final project will use. In particular, markdown-it-html5-media seems to be too restrictive in terms of what it will allow you to do, it won't allow you to use locally hosted videos and it forces you to write code to get around the usage restrictions. The terms of the license are also problematic (using the MIT license is not the same as forcing contributors to release their code to the public domain).

Next we have the block of code that generates HTML fragments from each

Markdown file in a directory we pass from the command line:

The first step is to capture the directory we want to work with from the command line using `proc.argv[2]. The parameters are zero based and they go in this order:

0. The name of the executable to run (Node)
1. The name of the script we want to run (index.js)
2. The path that we want to work with. We could also include a default using the logical OR operator

We read the directory using Node's fs.readdir to get a list of all the files in the directory we specified.

For each file in the directory, we read the file, render the markdown and store into a variable and write to a file using fs.writeFile

If readdir, readFile or writeFile causes an error we log it to the console and bail.

```
let dirToRead = proc.argv[2];

fs.readdir(dirToRead, 'utf-8', (err, files) => {
  if (err) {
    console.error(err)
    process.exit(-1)
  }

  files.forEach((file) => {
    let fullPath = dirToRead + file;
    fs.readFile(fullPath, 'utf8' , (err, content) => {
      if (err) {
        consol'utf8'r(err)
        process.exit(-1)
      }
      const processedFile = md.render(content);
      fs.writeFile(`src/fragments/${file}-fragment.html`, processedFile, (
        if (err) {
          console.error(err)
```

```
          process.exit(-1)
        }
        console.info(`${file} fragment saved successfully`);
      })
    })
  })
})
`${file} fragment saved successfully`
```

The next block is the template pieces we'll use to build the full page.

There is one string for the top of the document; everything until the opening body tag and one for the bottom of the document, the closing body and html tags.

I chose to use template literals to make my life easier. This way I don't have to escape characters or alternate quotation marks.

Another advantage is that I can directly insert script and stylesheets on the templates. We'll explore this in more detail when we look at FontFaceObserver and the scripts it uses.

```
const documentTop = `<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
`;

const documentBottom = `</body>
</html>`
```

The final piece is to assemble the full page from the templates and the HTML frements created from Markdown.

Rather than use readFile and writeFile, we'll use `fs.writeFileSync` to make sure that the file is new and catches our changes and `fs.appendFileSync` to append the content and the footer synchronously.

We read the directory where we stored our fragments (`fragmentSourceDir`) and error out if there was a problem.

For each file in the fragment directory we do the following:

We read the file and bail if there was a problem.

We append the `documentTop` template literal, the file content (since it's a fragment there's no extraneous markup to remove) and the `documentBottom` template literal.

```
fs.readdir(fragmentSourceDir, 'utf-8', (err, files) => {
  if (err) {
    console.error(err);
    process.exit(-1);
  }

  files.forEach((file) => {
    const fullPath = fragmentSourceDir + file;
    const destination = `${file}-full.html`;
    fs.readFile(fullPath, 'utf8', (err, content) => {
      if (err) {
        console.error(err);
        process.exit(-1);
      }

      fs.writeFileSync(`dist/${destination}`, documentTop, 'utf-8');
      fs.appendFileSync(`dist/${destination}`, content, 'utf-8'utf8'    fs

      console.log(`${destination} file created`)
    'utf-8'`dist/${destination}`
```

What's next

We have a working converter from Markdown to a complete HTML file. Is it

elegant? No, but it works.

This section will address some things to do moving forward from the MVP code we currently have. They may or may not include code at this stage, but they will before this project goes to 1.0

Fixing the name of the files as they are placed in their final location

One thing that botthers me is that I can't figure out is renaming the file. Out of convenience, the final name of a file is something like `hello.md-fragment.html-full.html`, there has to be some process or tool similar to gulp-rename, maybe with regular expressions.

Polishing the template and (maybe) moving them to separate files

The template literals can use a little love. If there are scripts or stylesheets that we need, we can add them directly to the literals.

Some examples of scripts and styles:

- Styles for the items that we already use, the fonts we want to work with and, possibly a basic layout
- Scripts and styles for Prism.js
- FontFaceObserver scripts

Example of template modification: FontFaceObserver

I've chosen to work with Recursive as my font. It's a variable font that handles Serif, Sans-serif and monospaced in a single font-file.

We need to load the font using @font-face making sure that we include the font-display descriptor to indicate how we want the web font to replace the default font.

and then use in the CSS. In this example, I've set as the font-family in the `html` element.

```
@font-face {
  font-family: Recursive;
  src: url(../fonts/Recursive_VF_1.078.woff2) format('woff2');
  font-display: swap;
}

'woff2'html {
  font-family: Recursive, sans-serif;
  font-size: 100%;
}
```

We then use FontFaceObserver to provide an additional fallback to improve font performance. I place these scripts on the footer to ensure that they won't block the rendering of the page. Would much rather have content with crappy fonts than no content at all.

We first load the FontFaceObserver script in its own tag.

We then use an inline script to define a FontFaceObserver object for our Recursive font (the name has to be identical to the name we defined in CSS).

We execute the `load()` method of FontFaceObserver and wait for the promise and log the result to the console.

```
<script src="js/fontfaceobserver.js"></script>
  <script></script>
    const font = new FontFaceObserver('Recursive');

    font.load().then(function () {
      console.log('Font is available');
    }, function () {
      console.log('Font is not available');
    });
  </script>
```

With these changes we have working HTML documents with additional functionality like FontFaceObserver and Prism.js.

It also provides extensibility and customizability to suit your needs. Taking the existing code as an starting point, I'll create multiple versions of the templates to address the optional steps in our requirements specifications.

The code is available on Github in a more complete and polished version.