# Building my own static site generator

In a way, this is the combination of two projects I've worked on at different times:

- A static site generator using Nunjucks
- A standalone Markup processor

After looking at the two projects, I figured out that the next logical step would be to combine them since they both do the same task in similar but slightly different ways.

The standalone markup generator uses Node built-in methods to write the components of the template and insert the converted HTML fragment into the output file.

The static site generator uses a Gulp build system to run the Markdown conversion using the Marked parser and the pushes the conversion into a Nunjuks template.

## Questions about the code

As I've written the code I've also been working in answering these questions. They have dictated the shape of the current code and will likely dictate the shape of future changes.

### What Markdown parser to use?

Since the two projects use different Markdown parsers the first question to ask is which one to use.

The easiest option would be to use Marked, the parser that is bundled with the Nunjucks Gulp plugins. However, Marked lacks a good plugin ecosystem, or I wasn't able to find one so I moved on to a different solution.

I chose to work with Markdown-it, the parser I used for the standalone project. It provides a rich plugin ecosystem and it makes it possible to write a lot more of your content in Markdown, without having to write HTML.

## Using Gulp to run the system

The next issue is whether we want to use a build system. Rather than pick up a new system to experiment with, I will use Gulp.

This allows me to leverage code from existing projects to build the necessary tasks for the generator and it also allows me to expand beyond just Markdown and take care of other tasks like Javascript, CSS, and image compression.

## Using Front Matter to customize items

One of the things that has always bothered me about generating content from templates is the impossibility of editing individual items like titles, metadata and other items specific to individual pages.

Front matter solves that problem. Using a combination of metadata and plugins allows you to customize individual items without having to do manual edits.

# Other than templates, what do we need?

I have a set of tasks to accomplish in addition to the Markdown conversion.

- Transpile Javascript with Babel
- Use SASS and Autoprefixer with PostCSS to transpile and add vendor prefixes to CSS
- Convert images to different formats and compress them to reduce file size using Libsquoosh
- Provide a preview server using Browsersync

# The code

Now we'll look at the code itself and break it into sections to make the narrative more coherent.

## Configuration and constants

The first block defines constant that we'll use throughout the script to make it easier on ourselves. If there's anything we need to change then we change it in one place and done.

```
// Nunjucks consts for file location
const dist = 'docs';
const src = 'src';
const templates = src + '/templates';
const content = src + '/pages';
```

The second block defines the Nunjuks environment we will use.

```
// Where to pull files from?
const env = new nunjucks.Environment(
  new nunjucks.FileSystemLoader(templates)
);
```

## Markdown

The first task uses Markdown-It to convert Markdown files to HTML fragments. The fragments are valid HTML files but are not full pages so they won't necessarily render correctly on a browser and have no styles of their own.

```
gulp.task('markdown', function() {
  const config = {
    options: {
      preset: 'commonmark',
      'commonmark'      xhtmlOut: true,
      linkify: true,
      typographer: true,
    },
  };
  return gulp
      .src('src/md-source/*.md')
      .pipe(markdown(config))
      .pipe(gulp.dest('src/html-source/'));
});
'src/md-source/*.md'
```

The second task uses custom code to create the template that will work with Nunjucks.

I created a custom solution because the tool I originally used, `nunjucks-markdown` did not work as intended. I was not able to convert the Markdown to HTML and get that automatically passed to the template so I added the extra step.

There are two string literals containing the top and the bottom of the template respectively.

The top template contains basic front matter and the two tags necessary for Nunjucks to process the file.

The bottom template contains the closing tag for Nunjucks to work.

```
gulp.task('assembleTemplate', function(done, layout = 'base') {
  'base'// string literal for the head of the regular HTML documentst docu
title: Default Title
description: Default Description
---

{% extends 'layouts/${layout}.njk' %}
{% block content %}

`;

  //'layouts/${layout}.njk'`---
title: Default Title
description: Default Description
---

{% extends 'layouts/${layout}.njk' %}
{% block content %}
```

```
`// string literal for the footer of the regular HTML document
 const documentBottom =
`{% endblock content %}`;

 const fragmentSourceDir = 'src/html-source/';
```

We read the directory containing the HTML fragments we processed with the `markdown` task.

Then for each file in the directory we:

1. Build the full path to the file by concatenating the directory and the filename. This is not the full path from root that we'd get from getting the full path from Node.
2. Read the file and exit if there is an error
3. Concatenate the top of the document template, the HTML content and the bottom template
4. Write the file to the file system

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

  files.forEach((file) => {
   const fullPath = fragmentSourceDir + file;
   const result = file.split('.html')[0];
   const destination = '.html'`${result}.njk`  fs.readFile(fullPath, 'u
    // if t'utf8'// if there's an error, log it to console and bail
    if (err) {
     console.error(err);
     process.exit(-1);
    }

    // Write the file
    const output = documentTop + content + documentBottom;
```

```
        fs.writeFileSync(`src/pages/${destination}`, output, (error) => {
      });
    });
  });
  done();
});
`src/pages/${destination}`
```

The final task is to actually render full pages based on the templates we just built.

We take all the template files that end with `njk` (Nunjucks) or `html` (for HTML templates), process it through `gulp-nunjucks-render` and put the result into the `docs` folder (we use `docs` rather than `dist` because `docs` is the default folder for Github Pages)

```
gulp.task('renderContent', function() {
  return gulp.src('./src/pages/**/*.+(html|'./src/pages/**/*.+(html|njk)'
        path: ['./src/templates'],
      }))
      // output files in app folder
      .pipe(gulp.dest('./docs'));
});
'./src/templates'// output files in app folder// output files in app folde
```

## Sass and Autoprefixer

To create CSS I use SASS and Autoprefixer to generate correct CSS without having to write the prefixed versions myself.

The first task runs SASS to generate the corresponding CSS. If you've been working with SASS for a while there are some changes th at may catch you by surprise as they did me.

Dart SASS is now the default version so you probably want to work with Dart SASS on your projects going forward.

In Dart SASS, synchronous methods work better than asyc methods, for now. So we make sure we run SASS in synchronous mode.

Remember to run SASS in expanded mode if you will do further proccessing, like we do with the `processCSS`.

```
gulp.task('sass', function() {
  return gulp.src('src/sass/**/*.s'src/sass/**/*.scss'ss.sync({
        outputStyle: 'expanded',
      }).on('error', sass.logError))
      .pipe(gulp.dest('./src/css'));
});
```

The `processCSS` task will take the CSS generated from SASS as the input and then run it through one or more tools to modify the final output.

Right now, the only tool we use is Autoprefixer to save myself from having to write any vendor prefix by hand.

```
gulp.task('processCSS', function() {
  const PROCESSORS = [
    autoprefixer(),
  ];
  return gulp.src('src/css/**/*.css')
  'src/css/**/*.css'ps.init())
      .pipe(postcss(PROCESSORS))
      .pipe(sourcemaps.write('.'))
      .pipe(gulp.dest('docs/css'));
});
```

## Javascript

I don’t expect to use this task often but it is nice to know that I have it available to transpile code into a version that will support all the 2017 and later features I need in my code.

```
gulp.task('babel', function() {
  return gulp.src('src/es6/**/*.js''src/es6/**/*.js'p.sourcemaps.init())
      .pipe(gulp.babel({
          presets: ['@babel/preset-env'],
        }))
        .pipe(gulp.sourcemaps.write('.'))
        .pipe('@babel/preset-env'pts/**/*'));
});
```

## Image Compression

Until not too long ago, I used Imagemin to compress all my images. It was tedious to get them to work and the plugin ecosystem has been known to have version-specific issues.

LibSquoosh is the CLI version of [Squoosh] and [gulp-libsquoosh] and it has none of Imagemin's drawbacks.

Libsquoosh provides all the encoder libraries as part of the package so there are no additional downloads and the codecs are built on [WebAssembly] so there's no dependency on Node-Gyp or other third party compilation tools.

Finally, it also provides acccess to newer image formats like JPEG-XL and AVIF out of the box.

```
gulp.task('compressImages', function() {
  return gulp.src(['src/images/**/*.{png,jpg,'src/images/**/*.{png,jpg,wel
            // console.log(src);
            const extname = path.extname(src.path);
  // console.log(src);// console.log(src);encodeOptions: squoosh.DefaultEr
            };

            if (extname === '.jpg') {
              options = {
                encodeOptions: {
                  jxl: {},
                  mozjpeg: {},
                },
              };
            }
            if (extname === '.png') {
              options = {
                encode'.jpg's: {
                  avif: {},
                },
                preprocessOptions: {
                  quant: {
                    enabled: true,
                    numColors: 16,
                  },
                },
              };
            }

            return options;
          }),
      )
      .pipe(gulp.dest('docs/images/'));
});
```

## Utility Functions

We have three utility functions to clean the installation to a default state and copy resources to the `docs` folder.

I choose to copy resources because we want to be flexible. Rather than change tasks everytime we decide to add something to the process. We could do the same thing with CSS but if I decide to add anything, it will be to the `processCSS` task, not adding additional tasks to the process.

```
gulp.task('clean', function() {
  return del([
    'docs/',
    'src'docs/'ource',
    'src/pages/**/*.html',
  ]);
});
```

```
gulp.task('copy:scripts', function() {
  'src/pages/**/*.html' 'src/scripts/**/*.js',
  ])
      .pipe(gulp.dest('docs/scripts'));
});

gulp.task('copy:fonts', function() {
  return gulp.src([
    'src/fonts'src/scripts/**/*.js'/**/*',
  ])
      .pipe(gulp.dest('docs/fonts'));
});
```

## Preview server

I might want to see the results of the process before I publish it so I set up a Browsersync-based preview browser.

It will pull the content from the docs directory and serve them as a website.

```
gulp.task('serve', function() {
  browserSync({
    port: 2509,
    notify: false,
    snippetOptions: {
      rule: {
        match: '<span id="browser-sync-binding"></span>',
        fn: (snippet) => {
          return snippet;
        },
      },
    },
    server: {
      baseDir: ['.tmp', 'docs'],
      middleware: [historyApiFallback()],
    },
  });
});
```

The default task pulls everything together. We run all the tasks we've defined so far in sequence, one after the other.

**For some reason running the default task will not write the final files to the target directory. To make sure we get the content, run `gulp renderContent` again.**

```
gulp.task('default', gulp.series(
    'clean',
    'sass',
    'processCSS',
    'renderContent',
    'compressImages',
    'copy:scripts',
    'copy:fonts',
));
```

# Future work

We have a basic static site generator working. It produces the type of code I want and it produces all the assets needed for the static site.

Yet there is still room for improvement and a lot of things I'd like to do with the code.

## Add Markdown-it Plugins

Markdown-it has a very nice plugin collection that allows you to enhance the output of the Markdown source without having to add HTML to it.

The standalone parser generator uses a set of these Markdown-it plugins and, in theory, it should be possible to add them to a Gulp plugin. I just need to figure out how to do it.

## Making The Output Prettier

Prettier is a tool that formats code according to parameters and specifications. We can leverage the Gulp-Prettier plugin to format the HTML output before writing it to file.

Need to do more research on how to do this. I believe it's a matter of configuration, but I'm not sure.

## Customizing Layouts

Right now I've created SASS and CSS files but I haven't done much work on the CSS side. I'd like to create a better base layout for the site.

This will also require a way to set up custom layouts for different types of pages. It might be possible to do it from the front matter but I'm not certain.

## Move From SASS to PostCSS

When I started using SASS it was awesome because it was easy to use and had a lot more features than CSS at the time.

Over the years, CSS has gained features that were once part of SASS and other preprocessors. Here's a rundown of my favorite changes:

- Nesting (upcomging) allows you to nest selectors and create more readble CSS
- Variables (using the @property syntax from Houdini) allow you to create and modify variables to use in your CSS. Because the CSS variables are live, you can change them and see immediate results
- Grid and Flexbox

PostCSS is a Javascript tool that allows you to use future features in your current codebase, just like Babel does for Javascript.

Changing the process to generate the final CSS means changing some of the code and deciding how much of SASS I want to emulate and how much am I comfortable discarding.

## Change Browserlists Versions

Right now I’m working with a default configuration for Browser List. This will tell the applicatioons that use it, like Babel, PostCSS and others, to only change what’s necessary for the supported versions.

The idea behind changing the browser list configuration is to make sure that we adhere to a sane configuration as described in Publish, ship, and install modern JavaScript for faster applications.

We’ll have to evaluate what impact, if any, the new browserlist has on our CSS and other items that use them.

## Properly Configure @babel/preset-env

@babel/preset-env provides a way to target specific browsers, either by using Browserslist’s list or by providing a list of browsers.

In the past I would have recommended switching to @babel/preset-modules because it potentially provides an even smaller file size by not rejecting the entire module if there are issues.

The features from `@babel/preset-modules` have been rolled into `@babel/preset-env` so we should be able to properly configure it to obtain the same result.

## Automating with Github Actions

The final bit I want to work with is to use Github Actions for building the site automatically when we push changes. I know it is possible to do it, I do this with my writing samples. But will it work with this project?