# Static Site Generators

I've looked at static site generators like Hugo, Gatsby, and Jekyll among others. They all have their strengths and weaknesses but they are overkill if all you want to do is throw together a quick prototype with a few pages and stylesheets.

# Markdown, HTML and templates: Version 1

Before we start we'll take the following steps:

- Create our root directory (`static-gen`)
- Create two working directories (`public` and `src`)
- Initialize `package.json`

```
mkdir -p static-gen/src
mkdir -p static-gen/public
npm init --yes
```

The first version uses the wrap-around system that I use to generate content for my blog. I've described the process in detail in Generating HTML and PDF from Markdown.

Install the packages we need to run the conversion tasks:

```
npm i -D  gulp@3.9.1 gulp-remarkable gulp-newer gulp-wrap
```

The two tasks that run the conversion are shown below. The first task converts the Markdown into an HTML fragment using the Remarkable markdown parser.

```
gulp.task('markdown', () => {
  return gulp.src('src/pages/*.md')
  'src/pages/*.md'wn({
        preset: 'commonmark',
        html: true,
```

```
        remarkableOptions: {
          html: true,
          typographer: true,
          linkify: true,
          breaks: false,
        },
      }))
      .pipe(gulp.dest('src/converted-md/'));
});
```

The second task inserts the resulting HTML into an HTML template that contains all the styles and scripts that we want to run on the pages.

```
gulp.task('build-template', ['markdown'], () => {
  ret'markdown'rc('./src/converted-md/*.md')
      .pipe(wrap({
        src: './src'./src/converted-md/*.md',
      }))
      .pipe(extReplace('.html'))
      .pipe(gulp.dest('docs/'));
});
```

This version has a problem: It keeps escaping the code and presenting it as a preformated code inside `pre` and `code` tags. for the templates to work with both Markdown and HTML we must handle template creation seprately for each format. These are still not full HTML pages but are written in HTML rather than Markdown so using the HTML extension is important.

The new template looks very similar to the one we're using to handle Markdown:

```
gulp.task('build-html-template', () => {
  return gulp.src('./src/pages/*.html')
      .pi'./src/pages/*.html'c: './src/templates/template.html',
      }))
      .pipe(gulp.dest('docs/'));
});
```

We've only covered the HTML generation portions of the template-based static site generator but it does more. Out of the box it will handle SCSS to css transpilation, ES2015+ to ES5 transpilation and image compression using Imagemin. Since it's Gulp-based you can integrate any other Gulp supported task into the process.

Because we're passing the results directly to the template we can add any type of HTML that we want, whether directly as HTML tags and attributes or Markdown to be interpreted.

## Future Evolutions

Right now all pages are converted using the same template. This works but it's inflexible. We could create additional templates and associated Gulp tasks to create different HTML based on the templates but it's not really productive. In the next post we will look at using a templating engine to generate our content.

# Using a templating engine by hand: Nunjucks

A next step is to build our own templating solution using Gulp. I've spent longer than I wanted in crafting this solution and it's still, like idea I based this from, has a lot of things I'm working on understanding and changing.

The idea of using a templating engine is to have options when creating content. We can create partial content blocks and define different layouts for our content.

Granted, this moves away from our simple model when using templates but we gain more flexibility in what we can do with the templated content.

We need to install the NPM packages that we need to make this work.

```
npm i -D nunjucks-markdown \
marked \
gulp-rename \
nunjucks\
gulp-nunjucks
```

We then require the packages we installed.

```
// Nunjucks and Markdown
const nunjucks = require('nunjucks');
const markdown = require('nunjucks-markdown');
const marked = require('marked');
const gulpnunjucks = require('gulp-nunjucks');
```

Rather than copy our directory paths in multiple places we write them down one and then reference them wherever we need them.

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

Here is also the first difference with our template copy. Rather than separate our partials (files with `.njk` as the extension) and our pages (`.html` files) we put them all in once place. The partials directory I used while developing this project looks like this.

```
partials
├── about.html
├── base.njk
├── css-containment.html
├── footer-scripts.njk
├── from-markdown-to-html.html
├── head.njk
├── index.html
├── javascript-dom.html
├── latex-to-web.html
└── voice-ui-agent.html
```

Next we use the Environment class and the FileSystemLoader to load templates from the specified directory; the one we defined in the `template` variable.

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

The next step is optional and configures the Marked Markdown parser. Because all assets are compiled at build time there is not as much worry about sanitizing the output of Marked.

**Do not change the `sanitize` setting if you will accept user templates or if you're using third party code.**

Once the configuration is complete we register the `marked` instance to work with `nunjucks-markdown`.

```
// Markdown options
marked.setOptions({
  renderer: new marked.Renderer(),
  gfm: true,
  tables: true,
  breaks: false,
  pedantic: false,
  sanitize: false,
  smartLists: true,
  smartypants: true,
});

markdown.register(env, marked);
```

Now that configuration is complete we can work with Gulp to create the rendering task.

We use all the HTML files in our `partials` directories and them compile them. This will take care of creating the fully templated HTML and converting any Markdown inside the pages to HTML. It will place the resulting pages inside the `docs` directory. I chose `docs` because it's one of the default directories that Github Pages allows you to use when documenting a repository.

```
gulp.task('pages', function() {
```

```
    return gulp.src([templates + '/*.html', templa'/*.html'*/*.html'])
        // Renders template with nunjucks and marked
        .pipe(gulpnunjucks.compile('', {env: env// Renders template with nun
```

That's it. We have a far more flexible structure to build from and we can create our own design system components to make out lives easier in the long run.

As with the templating solution we can create additional tasks to enhance the resulting HTML pages. I've created a proof of concept project that illustrates how this works.