Build a style guide

For the longest time I've resisted building pattern and component libraries for my projects. I've thought about them as a waste of time and resources. I conceptually understand their importance and how they benefit projects.

At An Event Apart last year I heard Brad Frost speak about Atomic Design and, based on my earlier research in Polymer and Web Components, I decided to take a look at design patterns again without web components.

Why a style Guide?

I'm also planning a full rebuild of my old labs website and decided to have a pattern library to make it easier to work in building the pages that I want to use. I started the design using Fractal and will probably continue to do so with the caveats we'll discuss as we go through the process and build components

Starting with Fractal

Fractal is a component build system created by Clearleft is a fairly straightforward way to create elements and start building more complex structures based on those elements.

The idea is that you can build as small a component as you want. Most styleguides and pattern libraries build buttons, links and color palettes.

I haven't gone that far down the rabbit hole, and perhaps I should. I'm playing with colors, strctures (Inverse color headers) but this should give me the chance to do so.

The most important thing for me is the ability to create mockups of the files and attach different stylesheets to test what the element looks like under different parameters.

Issues that I have with Fractal

If there is one issue I have with Fractal is that it's a completely different environment with its own CLI and no plugin to run fractal from within Gulp other

than using `gulp-exec` to run the fractal CLI.

Furthermore there is no easy way to rebuild the components into a static website. It's a manual rebuild and reload process.

Exploring Fractal

Fractal is a Node application. To install it run the following command:

```
npm install -g @frctl/fractal
```

This will install a `fractal` command line tool.

```
fractal --help
┌──────────────────────────────────────────────────────────┐
| Fractal interactive CLI                                  |
|──────────────────────────────────────────────────────────|
| - Use the help command to see all available commands.    |
| - Use the exit command to exit the app.                  |
|──────────────────────────────────────────────────────────|
| Powered by Fractal v1.1.4                                |
└──────────────────────────────────────────────────────────┘

fractal ➤
```

Core Concepts

Now that we've installed and run Fractal we'll explore some of the basic concepts. This is not an exhaustive reference, for ythat go to the Fractal Guide that covers this in a lot more detail. This document covers how I use the tool.

View templates

Fractal can use any template engine we want. By default it uses handlebars with additional extensions. A basic Handlebar template looks like this:

```
<div class="entry">
    <h1>{{ title }}</h1>
    <div class="body">
    {{ body }}
    </div>
</div>
```

This will take the `title` string from a configuration file or as passed to the element and use it inside the h1 element. It will do the same thing with the body content and the div in the body element.

We can create partial templates to build more comple layouts. For example we can create a page template, a header template and a footer-scripts template to hold different parts of our page.

The `header` template contains the head element and all its children. We use it because, other than the tilte, these elements are not likely to change. The template looks like this:

```
<head>
    <link rel="stylesheet" href="/styles/video-load.css">

    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width,minimum-scale=1,maximum-scale=1">

    <title>{{title}}</title>
</head>
```

We use the `footer-scripts` to add scripts that we want to add to all content pages of our site. It contains all the scripts that I use so it may get pruned and made into different versions depending on what the specific page needs. The default script is shown below:

```
<script async src="/scripts/lazy-load.js"></script>
<script async src="/scripts/vend</script>-with-locales.js"></script>
<script async src="/scripts/vendor/prism.js"></script>
```

```
<script async src="/scripts/vendor/fontfaceobserver.standalone.js"></scrip
</script></script>
<script src="/scripts/lazy-load-video.js"></script>
```

The final template is for our basic pages. It will use the two partial templates we've seen before and build the page. The default Handlebars way looks like this:

```
<html>
{{> @header }}
<body>
{{> body }}

{{> @footer-scripts }}
</body>
</html>
```

We can nest templates inside each other to make content even more moular. We could take our header and title element and make a template out of them. We'll look at this in more detail when we build a component.

If I'm working on a style guide then I will use them extensively. If I'm building the style guide as part of a larger project I may stick with standard handlebars or those available from the handlebars-helpers collection, I don't want to stay tied to Fractal any more than absolutely necessary, particularly if the project is for a client.

You can check the existing Fractal helpers in the documentation guide (Using Handlebars section). I will use the `render` helper to illustrate how they work.

```
<html>
{{render '@header' }}
<body>
{{> body }}

{{render '@footer-scripts' }}
</body>
</html>
```

The result should be the same but behind the scenes Fractal will do additional work with the rendered template. We don't need to know what it is, it just works :)

Context data

Context data can be used to populate parts of the template. The two most common ways to do it are to create a .json file or to use YAML front matter at the top of the page. The idea is that you don't need to manually give titles or do things with the template, for example we could create the following JSON file for a component may look like this:

```
{
  "context": {
    "title": "Level 1 Heading",
    "Level 1 Heading""author": "Aragorn, King of the West"
  }
}
```

We could also provide the same data at the document level using YAML. The JSON file converted to YAML front matter works like this. We can extend this with additional information and data to work with our templates.

```
---
context:
  title: "Level 1 Heading"
  author: "Aragorn, King of the West"
---
```

Configuration files

Components, documentation pages and collections can all have their own (optional) configuration files associated with them.

> In order to be recognised, configuration files must:

- Reside in the same directory as the item that they are configuring
- Have a file name in the format item-name.config.{js|json|yml} - for example button.config.json, patterns.config.js or changelog.config.yml

Some things to note:

- The javascript version (the one that exports a module) is not as strict as the JSON version in terms of quoting string attributes

```
module.exports = {
    title: "Base Layout",
    status: "prototype",
    context: {
        "title": "Click me!",
        "author": "Carlos"
    }
};
```

```
{
  "title": "Base Layou"Base Layout""status": "prototype",
  "context" "title": "Clic": "!",
    "author": "": ""author""author": "Carlos"
  }
}
```

Some configuration items will have their values inherited from upstream collections or their default settings if the values are not set in the item's configuration file directly.

This can also be thought of a cascade of configuration values from their default settings down through any nested collection configurations and into the item itself.

Naming & referencing

Fractal is a flat-file system, and makes use of some simple file and folder naming conventions to help it parse the file system and generate the underlying data model.

One of the main disadvantages of flat-file systems is that when one item references another via a path, moving any of those items inevitably results in those links breaking. So Fractal also supports a reference system, whereby items can use 'handles' instead of paths to link parts of the system together.

Unless told otherwise, Fractal will infer the name of a component or documentation page from its view template file name (or the parent directory for ‘compound’ components). It will then use this name (plus some other information) to generate a handle for the item. Handles are what will be used to reference that item elsewhere around your project.

> Names and handles are both ‘slug’ type strings, and will contain only lowercase, alphanumeric characters plus underscores and dashes.

The name will also be used to generate a default label and a title for the item. Labels are the text that will be used when the item is referenced in any navigation (for example in the web UI) and the title value is the text that will be used anywhere else a human-readable name for the item is required.

For a template in blockquote-large.hbs:

```
├── components
|   └── blockquote-large.hbs
```

The following labels and slugs will be generated:

- name: blockquote-large
- handle: blockquote-large
- label: Blockquote Large
- title: Blockquote Large

Statuses

Pages can have statuses associated with them.

Each status has a colour and a label that can be displayed in the web UI (and other places) to help people quickly understand the status of each component.

Fractal defines some default statuses, but you are free to define your own to suit the needs of your project, or customise the colours and labels associated with these statuses.

Building a component

In this example we'll build a header element with title and author children. We'll build two separate templates, the first one has the content of the template that we can style as needed.

```
  <h1>{{title}}</h1>

  <p>by</p>

  <h2>{{author}}
```

The second template takes the content template and inserts it into a header element. The cool thing is that we can start with as small templates as we want or need to acommodate atomic design principles

```
<header>
  {{> @head}}
</header>
```

Moving the components to Fractal

To move a component into Fractal do the following:

1. Create the Handlebars template
    - Make sure you've already created all the components that you'll use inside the current element
2. Create the configuration file for your component
3. Create the styles (note that these are for display only, the style should go into your SASS or CSS)
4. Preview.

Static sites for Rapid Prototyping

Now that we have a way to create components and preview them using Fractal we can look at how we can use the same handlebars templates to build a static web site.

Because I've used Gulp to build the rest of the site it's important to me that whatever site generator I work with also uses Gulp. Assemble does enough of the work that it's worth putting up with its idiosyncrasies.

First the bad. There used to be a Gulp plugin that would make Assembly work with Gulp. The plugin has been deprecated and people cautioned to work directly with Assemble.

This would be all well and good except for the fact that documentation sucks. That would be a minimal issue if there were Gulp examples that people (developers or users) shared with the community or if the documented best practices were illustrated with code.

That said I got it to work in Gulp 3.9 and I'll document the process and why I think this is a good way to build static sites.

Why Static?

Rather than spin up a new server or write a whole new app every time we want to test a layout or do a lot of complex things when prototyping layouts I've chosen to build the prototypes as static sites. With static sites I still use HTML, CSS and ES6 but don't need to fireup a full server to make the demos work.

Because I'm using Handlebars templates I can reuse as many templates as I need or use them to create specialized versions. For exampe, if I have a template for a header element, I can copy it and make it into an inverse header by just adding a CSS class and swapping the foreground and background color.

I can also build pages with data without having to use a database. Using JSON and Handlebars we can build data-driven content without a database. It doesn't work as well as PHP and other dynamic lanauges but we're not serving the site as

a dynamic application, I'm just using the data to build the files that will be statically served.

The final part is easy of serving the content. I can serve them locally from a laptop for a presentation, I can upload them and serve them from my DVS where I also host my Wordpress installations and, the most intriguing option, I can serve them from buckets in AWS and Google cloud.

We'll explore these options as we progress through creating the content. But first let's look at what we need to do to get Assemble up and running.

Building an Assemble Gulp workflow

Installing Assemble is fairly easy. We install 3 Gulp plugins: `assemble`, `gulp-extname` and `helper-markdown`.

`assemble` is the core Assemble toolkit.

`gulp-extname` provides a quick way to replace file extensions based on predetermined mappings.

`helper-markdown` converts Markdown to HTML using the Marked library create by the same author who created Assemble.

```
npm i --save-dev assemble gulp-extname helper-markdown
```

Once we've installed the plugins we can create the Gulp task. This is what will take the pages, build them using layouts which in turn will use one or more partials to build the resulting HTML page.

The Assemble purists may complain that the task we defined should be broken down into two tasks, one to load configuration and the other to actually run Assemble. They are correct, that's the right way to do it... but I'd rather keep most or all the configuration in the task that uses it and this has worked so far.

So let's configure Assemble to create our content.

We first configure Assemble by assigning it to a constant. I'm doing this outside the task because I may want to break this single task into multiple Assemble tasks that reuse the app and some of the elements we define below.

The first action we do inside the task is to tell assemble where to find partials, layouts and pages to build the content.

Next I provide a default layout. I do this to make sure the content will build even if I forget to specify the layout to use. Better to have the wrong layout than no layout at all.

Because I write all my content in Markdown I need to be sure that I use Asssemble's Markdown helper. It uses Marked.js under the hood so the result should be the same whether I run it through assemble or run it through my starter kit.

There are times when I want to use JSON to populate data in the templates rather than have to type it all by hand; `app.helper` tells Assemble where to find the JSON or YAML files to populate data with.

I then push the pages to the processing stream, render each file, run then through `extname` to change the extension from `hbs` to `html` and then save them in the destination file specified in `app.dest` (in this case the `_site` directory).

```
// 1. Setup Assemble
const app = assemble()

gulp.task('build', () => {
  /'build'ts the locat// 2. sets the locations of partials, layouts and pa
  app.layouts('src/templates/layouts/*.hbs')
  app.pages('src/templates/pages/**/*.hbs')
  // 3. Sets the default layout
  app.option('layout', 'base')
  // 4. Tells Assemble to use the 'src/templates/layouts/*.hbs''markdown'
  // 5. Sets the locations for data files
  app.data('src/templates/{pages,partials,data}'markdown'// 5. Sets the la
   'pages'// 6. Pushes "pages" collection into stream
  return app.toStream('pages')
    // 7. Render pages with default engine (handlebars)
    .pipe(app.renderFile())
    // 8. Map the source extension to the desired output
    .pipe(extname())
    // 9. specify your output location
```

```
    .pipe(app.dest('_site'))
})
```

So now we have a working task that will convert all Handlebars template into HTML. Now we can worry about creating the content.

Partials, pages and layouts

When we built the Assemble Gulp task we mentioned three different types of Handlebars content: `partials`, `pages` and `layouts`. We'll look at them in more detail.

If you're not familiar with the concepts we'll look at them in more detail.

Partials are small reusable components for your page. These can be as small as a header or a list of links and as large as a card component. We make them partials to make sure we can reuse them in different pages.

An example of a partial Handlebars template:

```
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,minimum-scale=1,max
```

Another type of partial holds smaller pieces of content like the header content we defined above:

```
<head>
  {{> head-content}}
</head>
```

We can go as granular as we need to with our partials. The basic question I ask when I build a partial is how many pages will I use it on, if it's a single page then I'd rather keep it as part of the layout, otherwise I'll create it as a partial.

Layouts are the structure of the page we want to build. It is also one of the places where we can add partials to build our content.

In the example below we use several Handlebars techniques to make our

template do what we want.

```
<html lang="en" dir="ltr">

<head>
    <title>
       <head>NDLEBARS0___
            ___HANDLEBARS1___
            ___HANDLEBARS2___ | Essays
        ___HANDLEBARS3___
             Essay
        ___HANDLEBARS4___
    </title>

    <link rel="stylesheet" href="styles/main.css"></title>HANDLEBARS5___
      ___HANDLEBARS6___
    ___HANDLEBARS7___
</head>

<body>

___HANDLEBARS8___
  ___HANDLEBARS9___
___HANDLEBARS10___


</head><!-- scripts used throughout most pages -->
___HANDLEBARS11___
    ___HANDLEBARS12___
___HANDLEBARS13___

</body>
</html>
```

In the head of the document we use the `if` builtin helper to customize the title of the page if the title attribute is present and provide a generic title otherwise.

The `head-block` and `bottom-script` partials use the block partial technique. According to the handlebar documentation:

> The normal behavior when attempting to render a partial that is not found is for the implementation to throw an error. If failover is desired instead, partials may be called using the block syntax.
>
> ```
> {{#> myPartial }}
>  Failover content
> {{/myPartial}}
> ```
>
> Which will render Failover content if the myPartial partial is not registered.

When only one person is working on the project it's ok to use regular templates but even then it might be good idea to code defensively and provide failover content in case you give things the wrong name or put them in the same directory. I've done that many times and got frustrated because the pages were not rendering as intended.

Pages hold the content and additional content related partials. This is the content counterpar to the layout structure. In this case we use several techniques to make the content work with the layout and be performant.

Be aware that this is not the full content.

```
---
title: Ghost in the shell review and analysis
author: Carlos Araya
layout: base
---

{{#>title-block--inverse}}{{/title-block--inverse}}

<div class="essay container">

**WARNING: Fanboy hat firmly on. We may disagree on specifics
but this is my vision.**

## In World General notes
```

```
In this video play special attention to the director's interview.

<div class="youtube-player" data-id="GpsfXLa2g-s"></div>

Kuze, his story and his relationship with the Major is
modified from Stand Alone Complex 2nd Gig. I have to admit, I
prefer the story as told in the TV show (2nd Gig, Episode 11:
Affection) but it's not quite the same story and it blends
itself well with the story and ideas as told in the film.

The story of why the Major become a cyborg is original to the movie,
even though it has similarities with 2nd Gig and Arise even if the
movie shows an entirely different story of her parents being killed
by terrorists. 2nd Gig has the major and Kuze being severely injured
in an accident as children and receiving some of the first fully
prosthetic bodies, Arise has the major being fully cyberized at
birth because of a chemical spill accident that killed her parents.

The Cyberization in the anime is much different than the cyber
enhancement in the movie and, in a way, it's closer to Ghost in
the Shell: Arise in terms of where the story is but not in how many
people have been fully converted to prosthetic bodies (most of them
are in the military and don't own their own cybernetic bodies, the
government does; Only after the second episode of Arise that the
major becomes the owner of her own body).
</div>
```

The page uses YAML Front Matter to indicate the title and the author as well as the layout to use.

The page uses a block partial for displaying the author and title in a way that will render it in inverse text (white text over black background rather than the default black over white).

I also write special tags for video using a lazy loader script that is defained in the layout of the page. One of the beauties of using Markdown is that we can write HTML where Markdown itself is not enough.

Using data to populate content

The last bit I want to discuss on this post is how to use JSON to populate content in a page. **This is no dynamic content**, every time we make changes to the data we need to recreate the page that uses it. This approach is particularly good for pages where the content doesn't change frequently like menus or big project lists like the one I've done below.

The page template is fairly simple. It (re)uses `head-content`, `title-block`, `footer-block` and `bottom-scripts` to create the oage structure.

The new item is the use of the #each helper to loop over content. In this case it will look for `project.json` and for every item in that file it will call the `project-single` partial

```
---
title: Project Archive
author: Carlos Araya
layout: base
---
<!-- projects.hbs -->
<html>
  <head>
    {{#>head-content}}{{/head-content}}
  </head>
  <body>
    {{#>title-block}}{{/title-block}}

    <div id="card-container">
      {{#each projects}}
        {{> project-single}}
      {{/each}}
    </div>

    {{#>footer-block}}{{/footer-block}}
    {{#>botttom-scripts}}{{/botttom-scripts}}
  </body>
</html>
```

`project-single` on its own doesn't do anything, it doesn't have a context to work from. But when it runs inside the projects page it will take the data from each item to populate that run of the template. Think of it as a Javascript For loop.

The CSS is already defined in an external file.

```
<!-- project-single.hbs -->
<div class="card">
  <div class="title">{{name}}</div>
  <div class="content">
    <div class="description">
      <p>{{description}}</p>
    </div>
    <div class="metadata">
      <p>Project Status: <strong>{{stage}}</strong></p>
      <p>Project Type: <strong>{{type}}</strong></p>
    </div>
    <div class="title">Project Notes</div>
    <div class="notes">
      <p>{{notes}}</p>
    </div>
    <div class="notes-title">Project Links</div>
    <div class="action">
      {{#if url.code}}
       <a href="{{url.code}}">Code</a>
      {{/if}}
      {{#if url.writeup}}
        <a href="{{url.writeup}}">Writing</a>
      {{/if}}
      {{#if url.other}}
        <a href="{{url.other}}">Other</a>
      {{/if}}
    </div>
  </div>
</div>
```

What’s next?

I work in a parallel process. I create the content first (partials, layout and pages) with a thought about styles and creative uses of CSS, animations and Javascript.

Some of my challenges with static websites is how much modularization we need versus how many CSS and Handlebaras files do we create? Right now my SASS process creates one monolithic stylesheet from multiple partials. In a static site I might want to break the styles in the same way that I break the pages.

Working on converting the site into a PWA. You can can create service workers (Gulp and workbox-sw) and web app manifests (Web App Manifest Generator) or you can bite the bullet and do everything manually. That’s your call.

Hosting the content will be the next problem to tackle. I’m inclined to look at Google Cloud or Firebase hosting to host the content. I love Google Cloud and I have the way to programmatically upload the content, the only downside is that it doesn’t support TLS and HTTPS.