Creating a Jekyll Image Gallery

olivierpieters.be/blog/2016/02/26/creating-a-jekyll-image-gallery

“A picture is worth a thousand words” – I was with this philosophy I knew I wanted to one day add a gallery to my website. Finally, that day has come.

The code and text below is mainly an explanation of what the different files in my example gallery do. Please check the repo if you want to have the bleeding edge version of all files. The final result of this tutorial should look like this:

{% include gallery-layout.html data=site.data.galleries.ghent-light-festival2 %}

Copyright notice: all images are licensed under CC BY-NC-SA 4.0

History and Context

I use the Jekyll static website generator’ to generate my website from Markdown and HTML snippets flavoured with SCSS/CSS and JavaScript. There are already a lot of plugins available to generate additional pages such as archives and a lot has been published on how to solve some common problems such as add a search bar. For galleries too, there is a plugin available, but it did not live up to my expectations. There is no dedicated image viewer and I like my galleries to be a bit denser with better integration of portrait images (as opposed to just making the entire container higher). As a result, I decided to create a custom workflow.

I started a Google search to find a suitable JavaScript library, since creating one myself seemed a bit complex/over the top. Google suggested very interesting stuff such as PhotoSwipe and chromatic.js. I really liked the thumbnail gallery of chromatic, but the gallery itself not so much. There was no option or for responsive design, captions, zoom etc. Those are things PhotoSwipe excels at.

After some more searching I also discovered lightGallery and Isotope. Although a thumbnail gallery with Isotope is not as great as the result with chromatic, I did really like the ease of use and fact that it is easily integrated with a full screen gallery viewer. This full screen gallery viewer than would have been PhotoSwipe if not for the *excessive* JavaScript I had to write to make responsive images work. In the end I decided to combine lightGallery with Isotope. There are no major issues associated with using either, except that this combination requires jQuery. In the future I might port the entire thing to an (almost) dependency free setup. I generally tend to avoid having to include jQuery in features. But I don’t have to really learn how to write good JavaScript for now, so that’s the upside to the current setup.

Installing the Dependencies

First off, we need to install both libraries. You can do that on your own by downloading the source code on GitHub, but using a dependency manager is much easier. I will use bower. So run:

`bower install jquery lightgallery isotope`

There also is a save option (`--save`) ensures these dependencies are saved to a `bower.json` file. I configured bower to install these dependencies into my `assets` folder. You can do this by setting `"directory" : "assets"` in your `.bowerrc` file. This is the workflow followed in the Github repo.

After installing the dependencies, make sure you are using the latest version of Jekyll too, since what will follow has only been tested on Jekyll 3.0.1.

Creating the Gallery Data

Jekyll users might be tempted to go for a custom collection. We will not use the Jekyll collection option to generate the galleries. Instead, we will use a general layout structure (that still leaves a lot of flexibility) in combination with a `.yml` file in the `_data` directory.

First, we will create the gallery data file in the `_data` directory. I will assume you are using an image management tool (such as Adobe Photoshop Lightroom) to generate sets of images of different sizes. After creating a set of e.g. 24MP, 12MP, 6MP, 3MP and 1MP pictures, we will feed all this data to a Python script and generate the YAML data. The Python script I use is available as a gist. It basically assumes all files use the same prefix and a different suffix. The suffix does not matter, since the script renames all files to include their dimensions. This will later come in handy when we link the HTML to our image assets. The script is a simple Python scripts that searches images and lists them in the YAML format. It is unable to set titles/captions and select a 'best image' for example. These attributes need to be set manually.

The output of the script looks like this:

```
picture_path: ghent-light-festival/part1
pictures:
- filename: Lichtfestival-0012
  original: Lichtfestival-0012-4000x6000.jpg
  sizes:
  - Lichtfestival-0012-1414x2121.jpg
  - Lichtfestival-0012-2828x4242.jpg
  thumbnail: Lichtfestival-0012-thumbnail.jpg
- filename: Lichtfestival-0003
  original: Lichtfestival-0003-6000x4000.jpg
  sizes:
  - Lichtfestival-0003-2121x1414.jpg
  thumbnail: Lichtfestival-0003-thumbnail.jpg
```

The Python script will automatically select the smallest file as the thumbnail file and select the largest as the original file. The (optional) element `sizes` contains all images that are used for responsive design. The original and thumbnail file can be the same file.

Files in `sizes` must be different from the thumbnail/original file. Note that this output cannot be used directly to generate the gallery; some additional settings are needed (see below).

The script will also work non-destructively: it you already have a `.yml` file, it will be loaded and new data will be added without loss of the (possibly incorrect) original data. Because of this non-destructive character, one can add additional images sizes after adding the captions etc. Note also that errors introduced by the user will not be fixed by the script.

To generate a valid data-file, you will need to select a ‘best image’ to use in the gallery preview (if applicable), add a gallery title (not really used, but might be useful) and check the picture path that contains the actual pictures. All listed pictures will be referenced relative to this path! An example:

```
picture_path: ghent-light-festival/part1
preview:
  filename: Lichtfestival-0058
  original: Lichtfestival-0058-6000x4000.jpg
  sizes:
  - Lichtfestival-0058-2121x1414.jpg
  - Lichtfestival-0058-2999x1999.jpg
  - Lichtfestival-0058-4242x2828.jpg
  thumbnail: Lichtfestival-0058-thumbnail.jpg
```

After setting the global options, you might want to add some captions to your pictures. You can add a title and/or a caption with `title:` and `caption:` options to a specific picture. An illustrative example:

```
- filename: Lichtfestival-0135
  original: Lichtfestival-0135-6000x4000.jpg
  sizes:
  - Lichtfestival-0135-2121x1414.jpg
  - Lichtfestival-0135-2999x1999.jpg
  - Lichtfestival-0135-4242x2828.jpg
  thumbnail: Lichtfestival-0135-thumbnail.jpg
  title: Sint-Jorisbrug
  caption: °RAINRAINRAIN° by Bram Lemaire
```

By now we have created the data file that will generate the gallery. Now it’s time to get into HTML/JS to display the actual gallery!

Creating the HTML Page

Gallery Page Layout File

Now it is time to write the actual gallery. We will start by defining a layout (`gallery.html`) in `_layouts` for the gallery. This layout should look like this and is very basic:

```
---
layout: default
support: [jquery, gallery]
---

<h1 class="gallery-title">{{ page.title }}</h1>

{{ content }}
```

The basic layout is very minimal since galleries can take very different forms. However, every gallery needs a title, so that will be added automatically. Notice we did not include anything from our files in the `_data` directory. We will add the actual galleries later on, because then we have the option to include several sub-galleries. This way we can subdivide the gallery per day for instance. This last requirement might seem a bit far-fetched, but consider the following: after a long trip, you might want to create a page/gallery per day and an overview page to wrap up the entire trip. Having the flexibility of a hierarchical structure makes things more structured and easier for readers.

Gallery Layout File

By now, we have our basic structure. We will also use a template for the individual (sub-)galleries. This file will be saved in the `_includes` directory. This file is called `gallery-layout.html` and looks like this:

```
<div id="image-gallery{% if include.id_number %}-{{ include.id_number }}{% endif
%}" class="image-gallery">
  <div id="gallery-sizer{% if include.id_number %}-{{ include.id_number }}{% endif
%}" class="gallery-sizer"></div>
  {% for picture in include.gallery.pictures %}
  <div class="image-wrapper">
    <a href="/assets/photography/{{ include.gallery.picture_path }}/{{
picture.original }}"
    data-responsive="{% for size in picture.sizes %} /assets/photography/{{
include.gallery.picture_path }}/{{ size }} {{ size | split: '-' | last | split:
'.' | first | split: 'x' | first | strip }}{% unless forloop.last %},{% endunless
%}{% endfor %}" class="image" {% if picture.title %} data-sub-html="<div
class='lg-toolbar caption'><h4>{{ picture.title | escape }}</h4>{% if
picture.caption %}<p>{{ picture.caption | escape }}</p>{% endif %}</div>"{% endif
%}>
      <img alt="{{ picture.title }}" src="/assets/photography/{{
include.gallery.picture_path }}/{{ picture.thumbnail }}" />
    </a>
  </div>
  {% endfor %}
</div>

<script>
$(document).ready(function() {
  var $gallery{% if include.id_number %}{{ include.id_number }}{% endif %} =
$("#image-gallery{% if include.id_number %}-{{ include.id_number }}{% endif
%}").lightGallery({
    thumbnail: false,
    selector: '.image'
  });
});

// init isotope
var $grid{% if include.id_number %}{{ include.id_number }}{% endif %} = $('#image-
gallery{% if include.id_number %}-{{ include.id_number }}{% endif %}').isotope({
  percentPosition: true,
  columnWidth: '#gallery-sizer{% if include.id_number %}-{{ include.id_number }}{%
endif %}',
  itemSelector: '.image-wrapper',
  layoutMode: "masonry"
});

// layout Isotope after each image loads
$grid{% if include.id_number %}{{ include.id_number }}{% endif
%}.imagesLoaded().progress( function() {
  $grid{% if include.id_number %}{{ include.id_number }}{% endif %}.masonry();
});

</script>
```

I admit it, this file is ugly. What is does however, is not that hard. For now, ignore the `{% if include.id_number %}-{{ include.id_number }}{% endif %}` code pieces. Then we end up with something that looks a lot simpler! First we define the gallery wrapper (`div` with `image-gallery` class). Second, we define a sizing element (`gallery-sizer`) that defines the size of a basic element in our grid. Next, we can start looping over the individual elements in the gallery (for-loop). The data is passed through the `pictures` variable or the gallery (and accessed through `include.gallery.pictures`). We also have to include a picture path that contains the image source directory (`picture_path`). After defining the basic picture elements (original in the `a` tags and thumbnail as picture in the `img` tag), we need to add the data to the `data-responsive` HTML attribute for the responsive design. Finally we instantiate the lightGallery gallery and Isotope grid layout. Also notice the (currently useless `image-wrapper`). This element is not useful for the example depicted here because we use padding for our elements (see CSS), but it might be useful if no padding is used. Check out the Isotope Issues on Github.

Now take the `id_number` variable into account. Because we include this into the various ids and JS variables, we are able to include multiple instances of `gallery-layout` in one page. This will not be used a lot, but it is useful to have this option.

But how to use the different layouts? Easy, just use the Liquid include tag:

```
---
layout: gallery
title: A Very Basic Example
support: [jquery, gallery]
---

At the end of our wonderful three week road trip at the West Coast of the US, we
spent about four days in the wonderful city of San Francisco. The city's well
known for the Golden Gate Bridge and it's fog, but has so much more up its sleeve!

{% include gallery-layout.html gallery=site.data.galleries.san-francisco %}
```

Adding additional content is done in exactly the same way as content is added to posts. Get creative!

Captions

Noticed the following piece of code in the `gallery-layout.html` file?

```
{% if picture.title %} data-sub-html="<div class='lg-toolbar caption'><h4>{{
picture.title | escape }}</h4>{% if picture.caption %}<p>{{ picture.caption |
escape }}</p>{% endif %}</div>"{% endif %}
```

In this snippet, we use a hack to hide the caption since captions are *not* hidden by default in lightGallery. To make them hide when the toolbare is hidden, we need to add the `lg-toolbar` class to the file. The library will then do the necessary to hide the caption. But an additional problem arises: the `lg-toolbar` class sets the CSS `position` property to the incorrect value of `fixed` . To fix this, we will use yet another class: the `caption` class. This class resets the `position` to the default value (`static`). See also the part on CSS for the exact code.

We have discussed how to create the gallery and different layouts with JS libraries, but we will also have to include/load these libraries when loading the current page. To this end, we will change the content of the HTML `head` tag. We need to include the following:

```
{% if page.support contains 'jquery' %}
<script src="/assets/jquery/dist/jquery.min.js" type="text/javascript"></script>
{% endif %}

{% if page.support contains 'gallery' %}
<link rel="stylesheet" href="/assets/lightgallery/dist/js/lightgallery-all.min.js"
/>
<link src="/assets/lightgallery/dist/css/lightgallery.min.css" />
<link rel="stylesheet" href="/assets/masonry/dist/masonry.pkgd.js" />
<link src="/css/gallery.css" />
{% endif %}
```

Notices the `support` variable in the YAML front matter of each gallery? This field ensures we include required JS/CSS files to display the content properly. But we only want to include these in pages that really need this. Hence the need for a special variable. Including the variable in the individual gallery pages and not into the gallery layout allows to include additional frameworks later on a per page basis.

Creating a Gallery Overview Page

The final thing to create is an overview page (the `index.html` page in the `photography` directory). This is a little trickier because we did not work with a collection. As a result we will have to manually list the data to the Liquid parser. This is a small sacrifice I was willing to make since one typically adds a gallery now an then, unlike the (possibly) vast amount of image data that is generated automatically by the Python script.

In the file `/photography/index.html` we write:

```
---
layout: default
title: Photography
permalink: /photography/
---

<h1>{{ page.title }}</h1>

<p>Below are two example galleries. The first gallery illustrates basic usage. The
second gallery illustrate how to include several image galleries into one entry to
create more complex structures and tell better stories.</p>

{% assign count = 0 %}
{% assign align = "left" %}
{% for gallery in site.data.galleries.overview %}
{% if count == 0 %}<div class="row">{% endif %}
  <div class="half-width gallery-preview {{ align }}">
    <h1>{{ gallery.title }}</h1>
    <a href="/photography/{{ gallery.directory }}.html">
      <img alt="{{ gallery.title }}" src="/assets/photography/{% if
gallery.picture_path %}{{ gallery.picture_path }}{% else %}{{ gallery.directory }}
{% endif %}/{{ gallery.preview.thumbnail }}" />
    </a>
  </div>
{% if count == 1 %}</div>{% endif %}
{% assign count = count | plus: 1 %}
{% assign align = "right" %}
{% if count >= 2 %}
{% assign align = "left" %}
{% assign count = 0 %}
{% endif %}
{% endfor %}

{% if count != 1 %}
</div>
{% endif %}
```

The design is very straightforward. To generate the gallery, we create a file in the data folder (`overview.yml`) that contains all needed data for the gallery overview. This makes handling the data easier since we do not have to modify the `index.html` every time. You can also include the data in the YAML front matter off course. Next, we loop over all galleries and add an entry. Because I want to have a two-column layout, I created the `count` and `align` variables (see CSS styling as well). These variables make sure the element styling is in order. We also need to make sure we close the `div` properly.

Adding Some Styling with SCSS

For the gallery specific styling, we create a `gallery.scss` file. This file is already included in the head above (last line). A basic example looks like this (don't forget to include the Jekyll front matter to make Jekyll compile the SCSS code):

```
$default-width: 50%;

* {
  -webkit-box-sizing: border-box;
     -moz-box-sizing: border-box;
          box-sizing: border-box;
}

/* force scrollbar, prevents initial gap */
html {
  overflow-y: scroll;
}

.image-wrapper,
.gallery-sizer {
  width: $default-width;
}

.image-gallery {

  display: block;

  .image-wrapper {
    float: left;
    padding: 3px;

    // hack required if no padding
    .image {
      width: 100.1%;
      width: calc( 100% + 1px );
      height: 100%;
    }
  }

  &:after {
    content: '';
    display: block;
    clear: both;
  }
}

.caption {
  position: static;
  color: #AAA;
}
```

Not all styling is required. The styling for the `.image` class for instance is redundant because we use padding between different images. Also check out this GitHub issue for more information.

To properly display the `index.html` page in the `photography/` folder, we need to include this in our css:

```
$on-palm: 600px;

.row {
    width: 100%;
    display: block;

    &:after{
        content: "";
        display: table;
        clear: both;
    }

    /* base definition */
    .column{
        float: left;
        display: block;
        box-sizing: border-box;
    }
    .full-width {
        @extend .column;
        width: 100%;
    }
    .half-width {
        @extend .column;
        @media screen and (min-width: $on-palm) {
            width: 48%;
        }
    }
    .one-third-width {
        @extend .column;
        width: 31.3333333%;
    }
    .one-tenth-width {
        @extend .column;
        width: 10%;
    }
    .nine-tenth-width {
        @extend .column;
        @media screen and (min-width: $on-palm) {
            width: 90%;
        }
    }
    .two-third-width {
        @extend .column;
        @media screen and (min-width: $on-palm) {
            width: 64.6666666%;
        }
    }
    @media screen and (min-width: $on-palm) {
        .left {
            margin-left: 0;
            margin-right: 2%;
        }
        .right {
            margin-left: 2%;
            margin-right: 0;
        }
    }
}
```

This can be added to the `main.scss` file or the `_layout.scss` file in the `_sass` directly. It really doesn't matter. This is included into every page because chances are you will need columns more often and we don't want to make it too complex with regard to the modular inclusions.

Example Gallery

This concludes this how-to. The entire how-to is applied to a default Jekyll blog that is available on GitHub. Checkout this repo for the exact file placement and to see the final result in action. If you find bugs, please contact me though mail (located in the footer) or open an issue.

Update: 28 July 2016

Due to recent changes in the lightGallery framework, the hack that was used for hiding captions might not function properly anymore. The issue can be solved using the following (less hacky) workaround.

First, change the `data-sub-html` attribute to the following:

```
data-sub-html="<div class='caption'><h4>{{ picture.title | escape }}</h4>{% if
picture.caption %}<p>{{ picture.caption | escape }}</p>{% endif %}</div>"
```

Then update the CSS to hide the element when the toolbars are hidden:

```
.lg-hide-items .lg-sub-html {
  display: none;
}
```

This will effectively remove the caption when the toolbar is hidden. Remark that this does not result into an animated transition between hidden en displayed. A possible way to animate it, is using the `visibility` and `transition` properties, as is visibility.