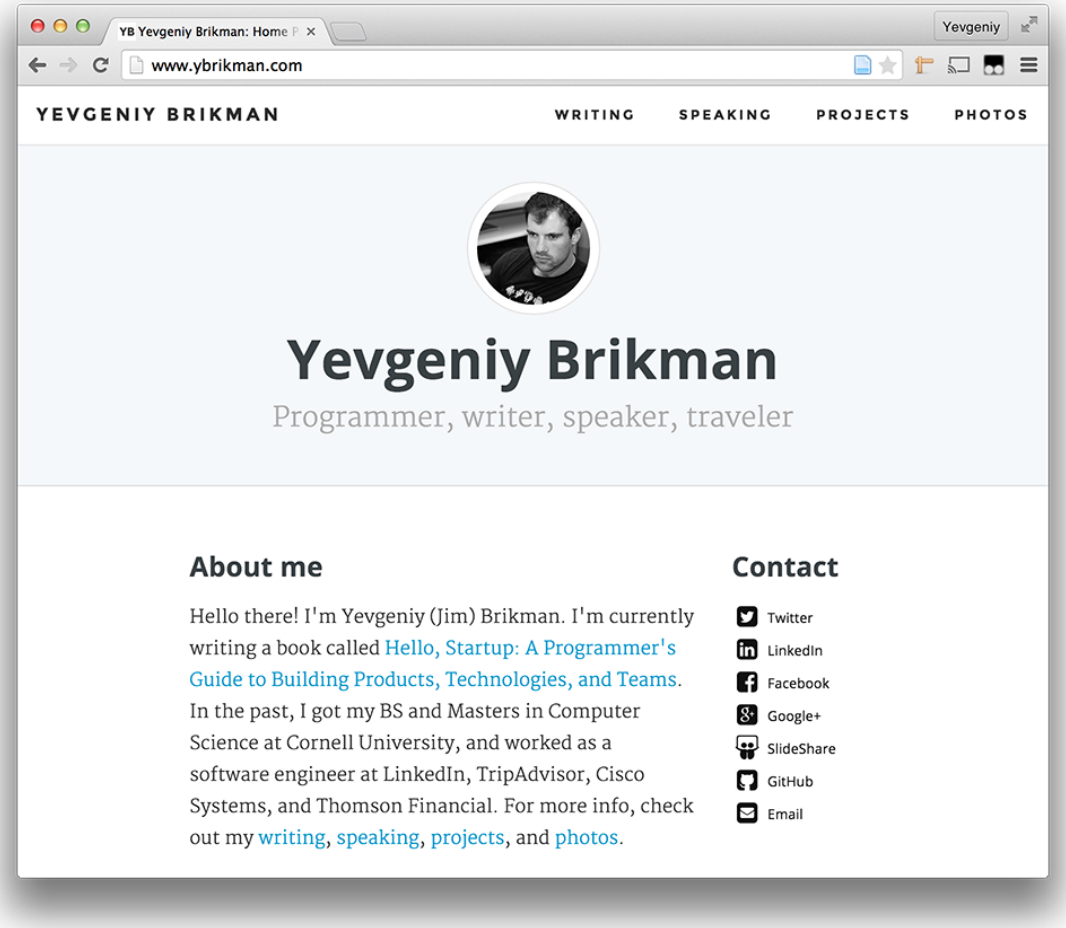
YEVGENIY BRIKMAN

Migrating from Blogger to GitHub Pages and launching the new ybrikman.com

Apr 20, 2015 Writing, HowTo 12 Comments 18 min read

Around 2007, I created my home page on a free PHP host and a blog on Blogger. The two websites have served me well for nearly 8 years, but they were long past due for an update. Last week, I finally sat down and rebuilt my home page on top of Jekyll, GitHub Pages, and Basscss, migrated my blog into it, and launched the new ybrikman.com:



I wish I could say it was a quick and painless process, but it wasn't. Coming up with a design was easy, but making it work in Internet Explorer and on mobile was painful. Learning Jekyll and GitHub Pages was easy, but the fact that GitHub Pages only supports a few Jekyll plugins was painful. Learning Basscss was easy, but filling in the UI elements it lacked was painful. And importing my posts from Blogger to Jekyll was easy, but making them look nice, and redirecting from Blogger to GitHub pages without losing SEO rank was painful. In short, just like every ground-up rewrite, it took much longer than I expected.

> *Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.*
>
> ***Douglas Hofstadter, Gödel, Escher, Bach***

As documentation for myself in the future, and as a guide for anyone else thinking of doing a similar migration, I've written this blog post to capture the key steps involved:

1. Design
2. Code
3. Migrate

Step 1: Design

In the spirit of dogfooding, I came up with a design by following the design chapter in my own book. One of the key lessons there is *design re-use*. Just as a programmer should prefer building on top of existing, battle-tested, and (preferably) open source libraries instead of reinventing the wheel, a designer should prefer using existing, battle-tested, and (preferably) open source designs instead of coming up with something from scratch. I browsed through my list of design resources, and found two free, open source templates to use as a starting point:

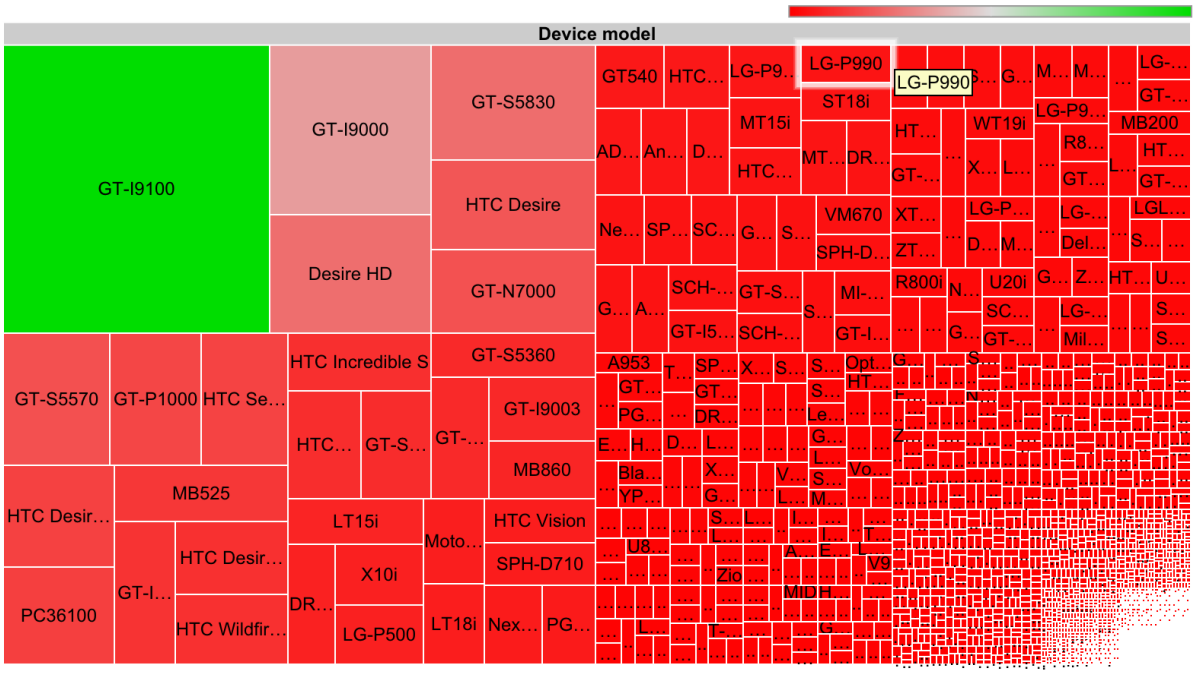
1. Kasper
2. Pixyll

I then added a menu based on the Freelancer Bootstrap Template, a comments and footer section inspired by Medium, and the simplest favicon I could think of (my initials) generated using favicon-generator.org. Copy and paste may strike some people as an uninspiring way to come up with a design, but the truth is that copy, transform, and combine are the basis of *all* creative work, as beautifully captured in the Everything is a Remix video series:

Everything is a Remix

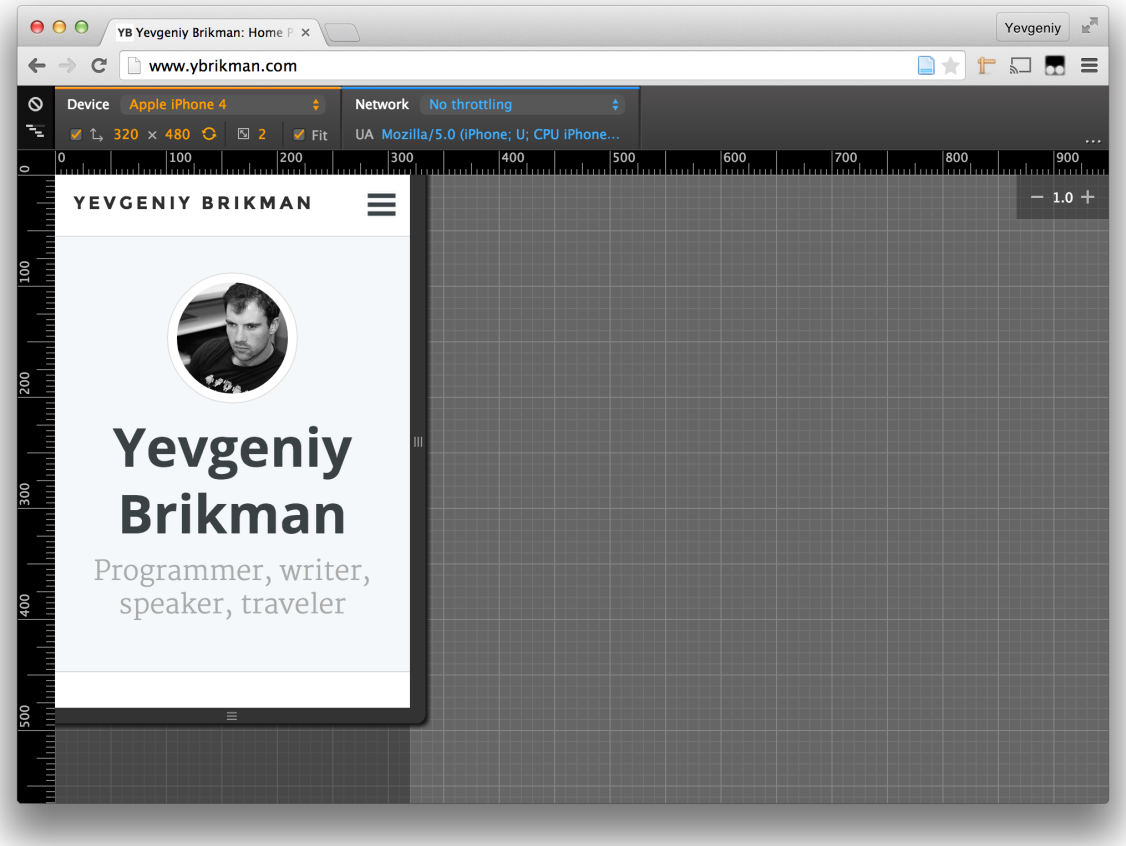
I ended up with a nice clean design, but once I started testing it in Internet Explorer (IE) and on mobile, I ran into a number of issues. Internet Explorer, at least the older versions, is problematic because, well, because IE is terrible:

Therefore, I've minimally tested my website in the latest IE, and everything less than version 10 is probably broken (which is OK, since that's less than 0.5% of my visitors). Mobile is tricky because of the insane number of screen sizes you have to support. For example, these are just the screen sizes for Android back in 2012:



Android fragmentation

The situation has only gotten worse in the last several years. The only two things that helped were a) learning some of the basics of responsive web design and b) making heavy use of the Device Mode and Emulation feature in Google Chrome:



Google Chrome Device Mode and Emulation

Step 2: Code

Let's now turn our attention to the code. The primary technologies I'll focus on are:

1. Jekyll
2. GitHub Pages
3. Styling
4. UI Components

Jekyll

In 2007, when I first signed up for Blogger, it was a fairly cutting-edge blogging platform. Today, it feels downright dated. It loads slowly, the available designs are uninspiring and cluttered, the templates are coded in an arcane XML syntax (more on that in the migrate section), and the in-browser writing and editing experience is awful (the WYSIWYG preview is inaccurate, it's hard to format things correctly, and several times, I've accidentally deleted something and it auto-saved, with no way to recover due to a lack of revision history). I played around with WordPress, Medium, and a number of other blogging platforms, but I found all of them too limiting, especially in terms of controlling the design and including arbitrary code. Therefore, instead of a hosted blogging platform, I decided to try to build my blog on top of Jekyll.

Jekyll is a static website generator. There are no databases, web frameworks, or complicated Content Management Systems (CMS) to deal with. You just create pages in HTML, Markdown, and Liquid, and Jekyll compiles them for you into a completely static website that consists of plain HTML files. You can think of it as a preprocessor for HTML in the same way that Sass is a preprocessor for CSS and CoffeeScript is a preprocessor for JavaScript. For example, to create a new blog post, you could put a markdown file such as `my-new-blog-post.md` into the `_posts` folder:

```
---
layout: post
title: Blog post title
---
```

```
This is my new blog post.
```

The section delimited by `---` at the top of `my-new-blog-post.md` is a block of YAML called the Front Matter that tells Jekyll to process this file. The `layout: post` line tells Jekyll to stitch the contents of `my-new-blog-post.md` into the layout defined in `_layouts/post.html`:

```
<html>
  <head>
    <title>{{ page.title }}</title>
    <link rel="stylesheet" href="/assets/css/main.css">
  </head>
  <body>
    <main>
      {{ content }}
    </main>
  </body>
</html>
```

The `{{ XXX }}` syntax is a variable lookup, so `{{ page.title }}` will be replaced by the title of the blog in your Front Matter ("Blog post title") and `{{ content }}` will be replaced by the contents of the actual blog post ("This is my new blog post"). Since you have full access to the Liquid template language, you can build your website with variables, layouts, includes, for loops, and lots of other features that keep your code DRY. And since the output is completely static, you can host it anywhere, such as Amazon S3, DropBox, or the host that I chose, GitHub Pages.

GitHub Pages

GitHub Pages allows you to deploy your code to the github.io domain (you can also set up a custom domain, as I did with ybrikman.com) just by pushing your code to a public GitHub repo. For example, the repo for ybrikman.com is https://github.com/brikis98/yevgeniy-brikman-

homepage, and every time I `git push` changes there, GitHub automatically runs Jekyll to compile the site, and deploys the new files.

The only downside is that GitHub Pages supports only a few Jekyll Plugins. You can use other Jekyll plugins on your local computer and check the compiled files in, but I've always found that error-prone, so for now, I'm trying to work around the lack of plugins as follows:

1. **Assets**: I'm using Jekyll's native support for sass to compile, concatenate, and minify my CSS. As for JavaScript, I'm just using a few open source libraries that are already minified and concatenated into one small file. If I end up with a lot more assets to manage in the future, I may switch to the jekyll-assets plugin and check in the compiled files instead.
2. **Tags**: using a small hack, I generated a single page with all of my tags. If I want to have a separate page for each tag, I'll switch to the jekyll-categories generator and check in the compiled files instead.
3. **RSS**: jekyll-rss-feeds works great with GitHub Pages to generate an RSS feed. No hacks required!

Styling

I use Bootstrap for most of my projects because it provides a great set of default styles, a nice grid system, and a large number of reusable UI elements and behaviors. However, I've never been a fan of how Bootstrap organizes its CSS and markup, and found that small changes often break my websites in unexpected ways. As a way to learn something new, I decided to try Basscss for my homepage redesign. Basscss is loosely based on Object Oriented CSS (OOCSS), which, in theory, makes your CSS more maintainable and reusable. For example, in the past, to put together the page that lists all of my blog posts, with a thumbnail on the left and the blog post description on the right, I might have written the following HTML:

```
<div class="blog-post">
  <div class="thumbnail">
    <img src="thumbnail.jpg">
  </div>
  <div class="description">
```

```
    <p>Description...</p>
  </div>
</div>
```

And I would have had the following CSS to go with it:

```
.blog-post {
  padding: 10px;
}

.blog-post .thumbnail {
  float: left;
  padding: 20px;
}

.blog-post .description {
  float: left;
  padding: 20px;
  font-size: 16px;
}
```

This CSS is not particularly reusable. It'll work for the list of blog posts, but not for other similar lists, such as my list of projects, which also has a thumbnail on the left and a description on the right. Of course, I could make the CSS class names more generic, such as replacing `.blog-post` with `.media-item`, but now the HTML and CSS for both pages is tightly coupled. What if the projects list needs to have more padding or use a different font? I'd have to insert all sorts of CSS overrides, which make it much harder to reason about and maintain the CSS.

The approach in Basscss is, as much as possible, to define small, granular, immutable (so you don't override them!), and therefore, highly reusable CSS classes. Here's a rough example of the CSS:

```
/* Reusable classes for padding */
.p1 { padding: 10px; }
.p2 { padding: 20px; }
```

```
.p3 { padding: 30px; }

/* Reusable float utilities */
.left { float: left; }
.right { float: right; }

/* Reusable font utilities */
.h1 { font-size: 28px; }
.h2 { font-size: 24px; }
.h3 { font-size: 20px; }
.h4 { font-size: 16px; }
```

I can toss these reusable CSS classes into my HTML as follows:

```
<div class="p1">
  <div class="left p2">
    <img src="thumbnail.jpg">
  </div>
  <div class="left p2 h4">
    <p>Description...</p>
  </div>
</div>
```

I could use the same markup in both the blog and project lists. If I wanted more padding in the projects list, I would change the `p1` to a `p2`; if I wanted a different font, I would change the `h4` to an `h3`; in either case, since the CSS classes are simple and immutable, I can be confident in making those changes because it's unlikely they will have any effect on the blog post list. In other words, this approach makes it *much* easier to reason about style changes, at least in the short time I've used it so far. The only downside is you end up with many more CSS classes in your markup, which does feel slightly verbose and repetitive, but the tools for dealing with this in markup (i.e. a templating language) are much better than in CSS, so it seems like a reasonable trade-off. Of course, what I've shown above is only a taste of how Basscss works (check out their design principles for more info), but so far, I'm pretty happy with the approach.

What I'm less happy with is that Basscss is a minimal and fairly low-level framework, so it does not have support built-in for nearly as many UI elements and behaviors as Bootstrap (side note: if you're looking for an open source project, I'd love to see Bootstrap rewritten on top of Basscss). As a result, I've had to do more work to find and glue together libraries that fill in the missing UI components.

UI Components

Here's a quick list of a few of other UI libraries I used:

1. responsive-nav.js: A nav that collapses down to the "hamburger icon" on mobile. Bootstrap has this built-in, and it turned out to be harder to replicate manually than I expected, so I turned to this library to take care of all the corner cases.
2. hover.css: Simple popover effects when you mouse over text or an image. Bootstrap has this built in as well, although this library does it with pure CSS, so no extra JavaScript is required.
3. lazysizes: A lazy loader that significantly speeds up page load time by only rendering images and iframes when (and if) they scroll into view. Very easy to use and it makes a huge difference in load time, especially for image and iframe heavy blog posts such as Must-See Tech Talks for Every Programmer.
4. Disqus: Add a few lines of JavaScript to your page and you have a full-fledged commenting system. It's free, fits well into most UI's, and has great moderation tools to fight spam.

Step 3: Migrate

Once I had the code for the website in place, the next steps were:

1. Import old blog posts
2. Redirect old blog posts

Import old blog posts

Jekyll has importers for a number of blogging engines, including Blogger. You download your Blogger posts as an XML feed and the Jekyll importer

converts them to static HTML files for you. This process doesn't take long, and it's certainly better than doing it by hand, but the results pull in a lot of the Blogger markup, which is verbose and includes a huge amount of inline CSS styles that break the look and feel of your blog. I wrote a *very* hacky Ruby script that stripped out as much of this markup as possible and converted the files to Markdown. I wouldn't recommend using it directly (yes, I committed the cardinal sin of using RegEx to parse HTML), but it might be helpful as a reference. Anything the script didn't handle correctly, I did by hand, which was a painful and time consuming process, even for my fairly small blog.

Redirect old blog posts

Now that I had all of my blog posts in Jekyll, the final step was to direct all traffic to the new URLs. I had two requirements:

1. **Redirect each blog post to its equivalent on my new website**. For example, if a user was looking at my old Blogger post, which has the format `http://brikis98.blogspot.com/2014/02/bar.html`, I couldn't just redirect them to `https://www.ybrikman.com/writing`, as they would never find the original `bar.html` post that they were looking at. Instead, I needed a way to redirect them to the equivalent Jekyll URL, which for me was of the form `https://www.ybrikman.com/writing/2014/02/03/bar`. Notice how transforming the URL from the old format to the new one requires a fair amount of string manipulation, including changing the domain name, adding the day of the year to the path, and removing the `.html` extension.
2. **Redirect without losing SEO rank**. This means I needed a way to redirect all search engine bots (so JavaScript probably won't work) and I needed a way to include a `<link rel="canonical" href="..."/>` tag on the page with the new URL.

I found some blog posts with instructions on migrating from Blogger to other blogging platforms, such as WordPress, that made it seem like this would be easy. It wasn't. Blogger templates use some sort of arcane XML syntax that only supports basic variable lookups, loops, and if/else statements, and I could not find any way to do the type of string manipulation I'd need to transform URLs as shown in requirement #1 above

(in fact, I found very little documentation in general for this XML syntax). I did come across a StackOverflow discussion that showed how you can do the string manipulation in JavaScript, but this fails requirement #2 above.

Therefore, it was time to build another hack. The idea was to generate an XML Blogger template that had all of my blog post URLs, and their equivalent redirect URLs, hard-coded into it as a series of if-statements. It would look something like this:

```
<b:if cond='data:blog.canonicalUrl == "http://brikis98.blogspot.com/2007/10/searc
  <link rel="canonical" href="https://www.ybrikman.com/writing/2007/10/31/search-
  <meta http-equiv="refresh" content="0; url=https://www.ybrikman.com/writing/200
<b:elseif cond='data:blog.canonicalUrl == "http://brikis98.blogspot.com/2007/11/i
  <link rel="canonical" href="https://www.ybrikman.com/writing/2007/11/01/i-hate-
  <meta http-equiv="refresh" content="0; url=https://www.ybrikman.com/writing/200
<b:elseif cond='data:blog.canonicalUrl == "http://brikis98.blogspot.com/2007/11/c
  <link rel="canonical" href="https://www.ybrikman.com/writing/2007/11/27/call-of
  <meta http-equiv="refresh" content="0; url=https://www.ybrikman.com/writing/200
<!-- And so on, one if-statement per blog post -->
```

I created another hacky Ruby script that ran through all the blog posts I converted earlier (see the import section above) and automatically generated this huge, hard-coded XML template for me. It's ugly, but now that it's in place, it seems to work. Or as the old saying goes, if it looks stupid but works, it ain't stupid.

The final step was to remove or update all the old Blogger URLs that I controlled. This included any links I had on social media (e.g. my Twitter profile, LinkedIn profile, Facebook profile, etc), as as well as any cross-references in the blog itself (I updated these automatically with yet another Ruby script).

Wrapping up

It took a fair bit of work, but I'm fairly happy with the results:

1. I completely control the look and feel of my blog and home page.
2. It loads quickly and works on mobile.
3. I can write blog posts in Markdown and include arbitrary code snippets.
4. I have full revision history powered by Git.
5. I get to use `git push` to deploy new content.

All the code for my blog is open source on GitHub under an MIT License, so feel free to fork it and reuse pieces as you wish.

Thoughts? Feedback? Leave a comment.

PREVIOUS

NEXT

A Guide to Hiring for your Startup

Yevgeniy Brikman

If you enjoyed this post, you may also like my books, *Hello, Startup* and *Terraform: Up & Running*. If you need help with DevOps or infrastructure, reach out to me at Gruntwork.

Share this post

Comments