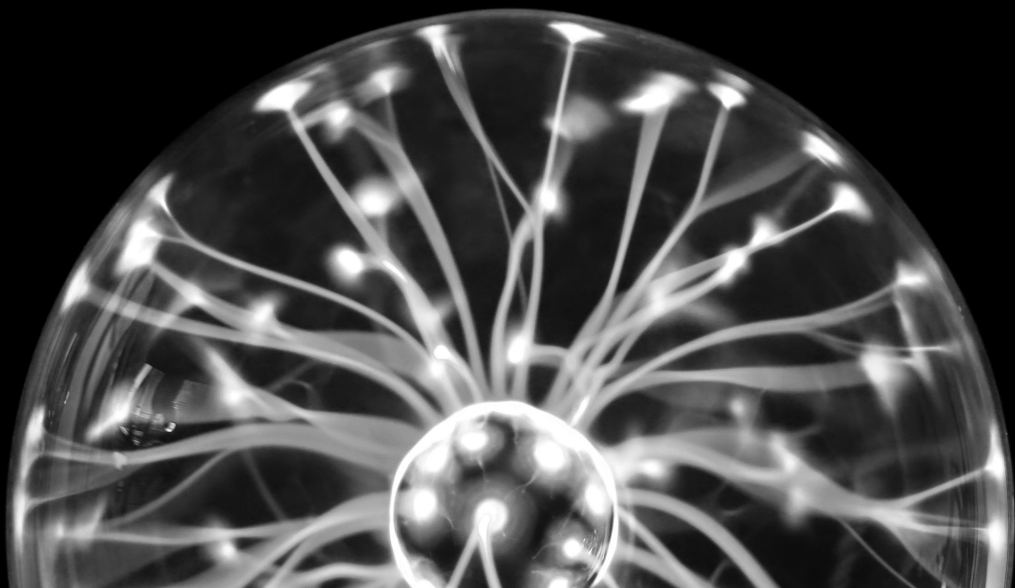# Static Site Generators

Modern Tools for Static Website Development

Brian Rinaldi

O'REILLY®

Additional Resources

3 Easy Ways to Learn More and Stay Current

Radar Blog Read more news and analysis about JavaScript, HTML5, CSS3, and other web platform technologies. *radar.oreilly.com*

Web Newsletter Get web development-related news and content delivered weekly to your inbox. *oreilly.com/web-platform/newsletter*

Fluent Conference Immerse yourself in learning at the annual O'Reilly Fluent Conference. Join developers, UX/UI designers, project teams, and a wide range of other people who work with web platform technologies to share experiences and expertise— and to learn what you need to know to stay competitive. *fluentconf.com*

Static Site Generators

Brian Rinaldi

Static Site Generators

by Brian Rinaldi

Editor: Allyson MacDonald	**Interior Designer:** David Futato
Production Editor: Matthew Hacker	**Cover Designer:** Randy Comer
Copyeditor: Amanda Kersey	**Illustrator:** Rebecca Demarest

September 2015:   First Edition

Revision History for the First Edition

2015-09-22:   First Release

Table of Contents

What Are Static Sites?

A Little Background

There's been a lot of talk recently about static sites and the new generation of tools used to create them, commonly referred to as "static site generators" or "static site engines." As with any new technology, it can sometimes be hard to differentiate the hype from the reality. This book aims to give you a broad understanding of the technology: what it is and where it best applies. First, however, we need to define what static sites are and where they came from.

The term "static site" is an interesting one if you think about it, as it defines itself by what it lacks. The "static" aspect doesn't so much describe a feature as the absence of one: dynamic page rendering. Once upon a time, probably before we commonly used the term "static site," this would have been considered a weakness.

Those of us who've been working in web development for some time probably recall building static sites using tools like Dreamweaver, HomeSite, or (heaven forbid) FrontPage. The content on these pages could only be changed by manually altering the existing site files and replacing the files on the server via FTP.

There were a number of issues with this process. Adding content to the site required a moderately high level of technical knowledge, either knowledge of the specific tool used to design and build the site or of HTML to handcode the site. One also needed to understand how to deploy the site to a host via FTP, which isn't necessarily straightforward for nontechnical users. This meant that the content creators, who are frequently nontechnical, could not directly or

easily contribute to the site and required the assistance of a web developer to add new content.

Creating new pages typically required copying and tweaking existing pages. As the site grew, maintaining proper navigation and links typically became both tedious and extremely error prone. Some tools offered features like templates that tried to solve these issues, but these could be complicated or cumbersome to create.

In addition to these issues, there was the limitation that if your site required dynamic features like comments or forums, for example, this was simply not possible in a purely static site.

The Dynamic Site Era

Dynamic sites seemed to fix these issues. Nontechnical content creators could create and update pages via backend forms without the need to understand the specifics of website development tools or HTML. Since the content and pages were all driven from a database, navigation could be generated automatically. In addition, by definition, dynamic sites allow for dynamic features such as forums or comments.

In the case of content-focused web pages, dynamic sites often took the form of a content management system (CMS). These could be custom built to the needs of the site or, very frequently, selected from a number of commercial or open source options.

To this day, most of the content published on the Web runs through some form of content management system. Popular open source options include Drupal, Joomla, and Typo3 (see Figure 1-1). Nowadays, these systems typically handle much more than simply content creation and publication, with features such as complex roles and access control, workflow management, document management, and syndication.

Figure 1-1. Adding an article in the Drupal CMS (source: Drupal.org).

These additional features lead to the biggest issue with dynamic sites, which is that the solution is often more complex than the problem. By virtue of its need to cater to a broad set of customers, a pre-built CMS often has a steep learning curve for both developers and content creators. Meanwhile, a custom CMS requires both extensive development efforts and access to a developer should issues or necessary changes arise.

Hosting dynamic sites is complicated by the need for database storage (and backups) as well as support for whatever dynamic language the site is built upon (PHP, Ruby, etc.). Factor in the need for regular updates to the dynamic language, database solution and even the CMS software itself, and it becomes rather obvious that, while dynamic sites solve many difficult problems, they bring with them their own set of complications.

The Rise of Blog Engines

The complexity of content management systems was not well suited for smaller, content-focused sites or blogs that didn't require advanced features like complex user roles or workflow. Blogging engines, the most popular being Wordpress (see Figure 1-2), aimed to solve this by making development simple, with pre-built and

easily customizable templates, and publishing content quick and easy.

Blog engines don't negate the need for supporting a dynamic language (PHP in the case of WordPress) or for a database (typically MySQL for WordPress). WordPress, however, became popular enough that many hosts made "out-of-the-box" hosting solutions that simplified setup and maintenance. To give you a sense of the popularity of WordPress, according to W3Techs, as of May 2015, Wordpress is used on approximately 23.9% of the top 10 million sites, a percentage that dwarfs every other content management system.
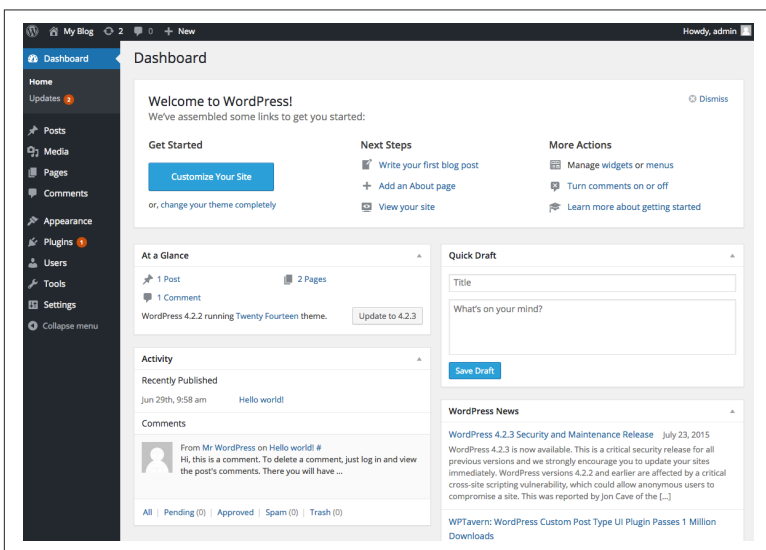
Figure 1-2. The WordPress dashboard (source: WordPress.org).

Nonetheless, over time, WordPress has begun to gain some of the complexity of a typical CMS, and it is generally lumped in the category of CMS by most industry research. Many sites heavily depend on features that are added via plug-ins, the quantity and quality of which can dramatically impact site performance. In addition, features like plug-ins and "shortcodes" can impact the portability of content, keeping your site tied to the Wordpress platform.

Some in the blogging community felt that Wordpress and competing blog engines like Moveable Type had strayed so far from the simplicity of their initial blogging focus that they created new

projects, like Ghost for example (see Figure 1-3), that aimed to get back to the basics of just blogging. Ghost's tagline is, in fact, "Just a blogging platform."
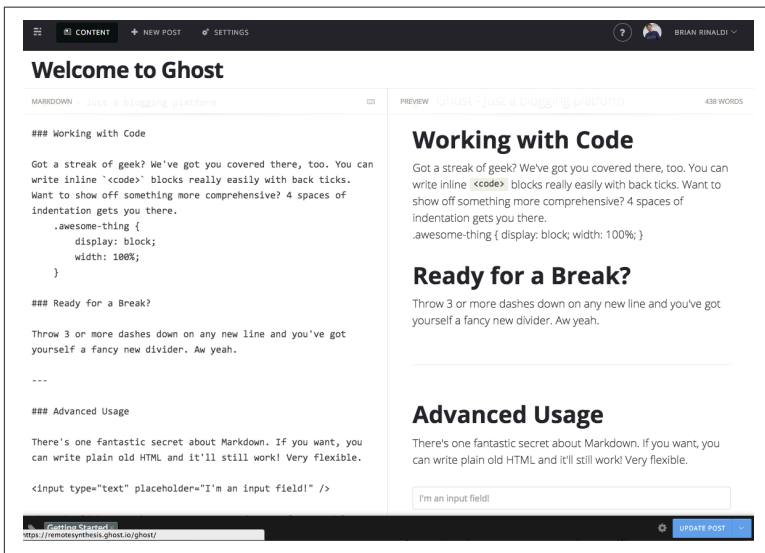
Figure 1-3. Ghost offers an intentionally simple and sparse editor (source: Ghost.org).

Static Pages Get New Life

Whatever complexity dynamic sites may bring, for most use cases, there is simply no avoiding the need for dynamic data. Even the most basic content site, like a personal blog, generally has dynamic aspects: commenting, feedback or contact forms and search, to name just a few. So it wasn't until the rise of new services that can fill these voids that static sites really became a viable option for more than just "brochureware".

There are numerous services, both free and paid, that offer the ability to add dynamic aspects into static pages (it's important to note that these services are not specifically intended for use only on static sites). Some popular options include:

- Disqus, Livefyre, or Facebook for comments
- Wufoo or Google for forms
- Google, Swiftype, or AddSearch for search
- Discourse for forums

There are many more covering a full range of typical site require-
ments. There even BaaS (backend as a service) solutions like
Parse or Kinvey that offer APIs that allow developers to pull any
form of arbitrary dynamic data into a static page.

> **NOTE**
>
> **Overview of Popular Services**
>
> If you're interested in some of the services listed above
> as well as implementation details, Raymond Cam-
> den wrote an article on the topic called "Moving to
> Static and Keeping Your Toys".

What makes all of these services work is the ability to load remote
data requests via Ajax. As an example, let's look at how to load Dis-
qus comments onto a page. The following is from my personal blog:

```
<div id="disqus_thread"></div>
<script type="text/javascript">
    /* * * CONFIGURATION VARIABLES: EDIT BEFORE PASTING INTO
YOUR WEBPAGE * * */
     var disqus_shortname = 'remotesynthesis'; // required:
replace example with your forum shortname

    /* * * DON'T EDIT BELOW THIS LINE * * */
    (function() {
        var dsq = document.createElement('script'); dsq.type =
'text/javascript'; dsq.async = true;
            dsq.src = '//' + disqus_shortname + '.disqus.com/
embed.js';
            (document.getElementsByTagName('head')[0] || docu
ment.getElementsByTagName('body')[0]).appendChild(dsq);
    })();
</script>
<noscript>Please    enable    JavaScript    to    view    the    <a
href="https://disqus.com/?ref_noscript">comments   powered   by
Disqus.</a></noscript>
```

In a nutshell, the script creates a new `<script>` element whose
source is a JavaScript file on the Disqus server. The file URL is spe-
cific to the forum via the configuration variable, `disqus_shortname`,
allowing it to retrieve the forum name via the URL of the
script. This file then performs a number of actions to remotely
retrieve comment data and display it on the page.

NOTE **How Disqus Works**

If you're curious for a more specific description, see "How does Disqus work?" in the Disqus documentation.

Of course, one need not rely on these services for loading dynamic data onto a static page—a savvy developer could write his or her own solution using similar techniques—but these out-of-the-box services make static pages a much more appealing, and far less daunting, option than they once were.

Defining a Static Website

So far we've covered some background showing how the static web pages of old failed to meet the needs of the Web as websites became more complex and interactive. We discussed how dynamic sites generally and content management systems specifically solved some of these problems but led to increased complexity in both development and authoring. Blog engines partially addressed these issues but also took on some of complexity over time. Finally, we saw how Ajax and the rise of services have helped make static pages a viable option again.

However, before we explore static site generators, I'd like to end our current discussion by laying a clear definition of a static site. Understanding what a static site is (and isn't) is essential for evaluating whether a static site generator is a workable solution for your project:

Static site files are delivered to the end user exactly as they are on the server.

> This is probably the key defining characteristic of a static site and part of why static sites tend to perform so well: there is no server-side generation at runtime. This means, for instance, that every visitor to your static site will be served an identical copy of *index.html* from the server until it is manually overwritten, say by uploading a new file via FTP.

There is no server-side language.

> It follows from the preceding characteristic that there would be no server-side language (like Ruby or PHP for example) involved. However, when speaking of static site generators,

some are written using these languages but are intended to be run locally.

There is no database.

As there is no server-side language to speak to a database, there is therefore no database. This does not mean that there is no data. There can be data stored as files or via an external service like the ones discussed earlier. This means that if you need common features like user registration/login, this would need to be via an external service.

Static sites are HTML, CSS, and JavaScript.

This seems fairly obvious, but it should be clear that since static sites are intended to run in the browser, they must rely on web technologies to function. Of course, this can also include images like JPEG and GIF, graphic files like SVG and WebGL, or data formats like JSON or XML.

Benefits of Static Sites

While each of the preceding features brings with it certain limitations, they also lead to some of the primary benefits of static sites:

Performance

There is no server-side processing and no database to connect to, meaning that there is nothing to slow down getting a static page from the server to your end user. This also means that there are no bottlenecks that might cause slowness or outages should you encounter a significant traffic surge.

Hosting

Since no server-side language is required, hosting requires no complicated setup or maintenance, making it cheap and easy. In fact, there are even free options, like GitHub pages or Surge, for instance (we'll explore deployment options in a later chapter).

Security

There are no server-side language issues to exploit and no database to hack. Basically, as long as the files on your host are secure, your static site is secure.

Content versioning

Since your entire site, from configuration to content, is file-based, it is very easy to keep all aspects of it within a version

control system like Git. This can be especially advantageous for things like documentation that you may want to allow community contributions, for example, using pull requests on GitHub.

Despite these benefits, static sites, even with the help of a static site generator, are not the solution for every type of site. In upcoming chapters, we'll discuss more some of the limitations of static sites and the types of sites these solutions are best suited for.

A Word (or More) About Markdown

Before we dig into static site generators, there's one last item we need to discuss: Markdown. Markdown has become a de facto part of the static site stack. It is a shorthand way to write HTML and is the default tool to write post and page content in most static site generators. However, it is often unfamiliar to most anyone who isn't a web developer.

What is Markdown?

Markdown is essentially a syntax for a simple, easy-to-read, plain text format that is designed to be converted to HTML. It was originally created in 2004 by John Gruber, who is well known for his commentary on the technology industry, and he owns the copyright as well as rights to the name Markdown, though the original conversion tool is licensed under the BSD open source license.

Markdown has been widely adopted across the industry as a way to quickly create web content using a simple shorthand. Many popular web-development tools offer Markdown support out of the box, including Sublime Text, Atom, Visual Studio Code, and Brackets. Most blog engines have started offering support for Markdown, including Wordpress.

There's even a burgeoning market for standalone Markdown editors, with some popular options being Mou on Mac, MarkdownPad on Windows, and Dillinger in the browser. Markdown support is also central to new services like Beegit, which offers online document collaboration.

More Markdown Tools

If you are interested in the tool ecosystem in Markdown, I wrote a post that covers more standalone options as well as conversion tools for doing tasks like converting Word documents to Markdown.

Markdown syntax

Markdown's appeal is the simplicity of its syntax. Its philosophy emphasizes being easy to read first and and easy to write second. Let's look at some examples to see how this works.

Headers are generally indicated using the pound symbol. So:

```
#My Title
```

results in:

```
<h1>My Title</h1>
```

And:

```
##My Header
```

results in:

```
<h2>My Header</h2>
```

The number of pound symbols indicates the header level. Markdown often offers multiple syntax options for elements, so headers can also be indicated via underlining. The following would also result in an <h1> block:

```
My Title
=============
```

Unordered lists can be created using either asterisks, pluses or hyphens:

```
* My first bullet
* My second bullet
```

results in:

```
<ul>
    <li>My first bullet</li>
    <li>My second bullet</li>
</ul>
```

Replacing the * with + or - will result in the same HTML output.

Ordered lists use numbers but do not require that the number actually correlate to the items position in the list. So:

```
1. My first item
1. My second item
8. My third item
```

results in:

```
<ol>
    <li>My first item</li>
    <li>My second item</li>
    <li>My third item</li>
</ol>
```

Italic and bold text typically also uses the asterisk, but can also use underscore. So:

```
*This is italic* and _this is italic_
but **this is bold** and __this is bold__
```

results in:

```
<em>This is italic</em> and <em>this is italic</em>
but <strong>this is bold</strong> and <strong>this is bold</strong>
```

Links and images use a similar syntax, one that my experience has found to be the least intuitive of Markdown's shorthand syntax. So:

```
![O'Reilly  logo](http://cdn.oreillystatic.com/images/sitewide-headers/ml-header-home-blinking.gif)

And this would be a [link to O'Reilly](http://oreilly.com)
```

results in:

```
<img src="http://cdn.oreillystatic.com/images/sitewide-headers/ml-header-home-blinking.gif" alt="O'Reilly logo">

<p>And this would be a <a href="http://oreilly.com">link to O'Reilly</a></p>
```

Hopefully this gives you a sense of what the Markdown syntax looks like. There is also shorthand syntax for things like block quotes, code blocks, and horizontal rules. If you would like a comprehensive overview of the entire syntax, refer to John Gruber's original syntax documentation.

The problem(s) with Markdown

Markdown's biggest flaw is the simplicity of its syntax. Once you become comfortable with the syntax, it can be very quick and easy

to write Markdown documents. But Markdown's syntax only covers a limited subset of HTML. To fix this limitation, Markdown allows you to directly include HTML within a Markdown document, but this means that you'll need to know HTML to properly use Markdown for authoring. There are also multiple "flavors" of Markdown to deal with. These issues can complicate using Markdown with content contributors. The following are two other problems:

Problem 1: the lack of a standard

There are numerous Markdown variations, called "flavors," available. GitHub relies on Markdown as a standard for its documentation and uses GitHub-Flavored Markdown. StackOverflow has its own additions to Markdown. According to Wikipedia, other variations of Markdown also exist from reddit, Diaspora, OpenStreetMap, and SourceForge.

There was even an attempt to standardize Markdown which ran into copyright issues, as John Gruber owned the rights to the Markdown name. It now exists under the name CommonMark.

The problem with a lack of a standard is that much of the tooling around Markdown is built for one variant or another. Some support multiple variants, but trying to teach a nontechnical content contributor about the complexity of the Markdown ecosystem can become a barrier.

Problem 2: Markdown doesn't replace HTML

Markdown covers a very limited subset of HTML, which means that authors will need to understand the situations that aren't covered as well as know the HTML to use for those situations. This forces a content contributor to not only learn Markdown, but also what Markdown cannot do, and then learn HTML to fill those gaps.

Let's look at a very common example. Markdown currently has no syntax for named anchors, but named anchors are frequently used in content to allow a user to quickly jump to a location in a page. In order to achieve a named anchor, you'll need to mix Markdown and HTML as follows:

```
<a name="mysubheader"></a>
##My Subheader
```

While Markdown's support for embedded HTML means that there is nothing HTML can do that Markdown cannot, it adds a

great deal of complexity, especially for a content contributor who is unfamiliar with HTML.

In addition, standalone Markdown editors are not WYSIWIG, opting instead to offer a live preview of hand-written code. As Markdown continues to grow in use, the tools keep improving, but the current state of Markdown tooling offers a very unfamiliar experience for many content contributors.

NOTE

Word to Markdown

One option for content contributors familiar with working in Word for content authoring is the Microsoft Word to Markdown Converter project by Ben Balter. My own personal use of this project has shown that while the output needs manual cleaning, it is generally reliable.

Despite these issues, as we'll see when we look deeper at static site generators, Markdown has become the standard for writing content within these tools.

The Basics of Static Site Generators

What Are Static Site Generators?

The basic concept of a static site generator (aka static site engine) is simple: take dynamic content and data and generate static HTML/JavaScript/CSS files that can be deployed to the server. This idea isn't new. The oldest static site generator tracked by the Static Site Generators list, which currently tracks 394 different projects, is 13 years old. Even some CMS systems have functioned this way for many years.

> **NOTE**
>
> **Comprehensive Lists of Static Site Generators**
>
> The Static Site Generators list is definitely the most comprehensive list of these tools available. While a list of nearly 400 tools may seem overwhelming, it offers helpful details, such as the language the tool is built with, when it was created, and even the last time it was updated, all of which you can sort by.
>
> StaticGen doesn't aim to be nearly as comprehensive, but it offers a good amount of additional details about each project. For example, it pulls the short and long description from the project repository and lists the templating languages it supports. You can filter by the language the tool is built with and sort by stars, forks, issues, or title.

Still, the new generation of static site generator tools tend to have some characteristics in common:

- Run via the command line
- Support one or more templating languages for theming
- Have a local development server for testing and debugging
- Support file-based data formats
- Have an extensible architecture
- Have a build process

Let's start by taking a look at each of these characteristics in a bit more detail.

Popular Generators and What They're Built In

Generator	Language
Jekyll	Ruby
Middleman	Ruby
Hexo	JavaScript
Pelican	Python
Hugo	Go
Wintersmith	CoffeeScript
Harp	JavaScript

Run Via the Command Line
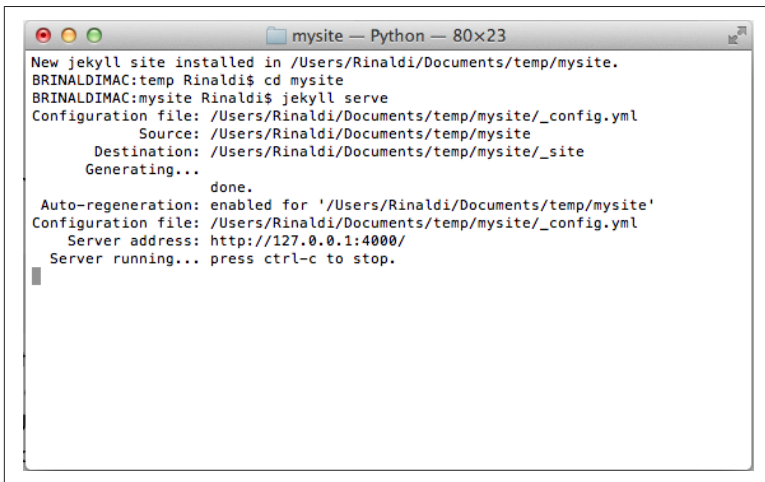
Most modern static site generators are designed to be run via a command-line interface, meaning that there is no GUI (graphical user interface). Everything from generating the default site files to running a test server to building and often even deploying the site happens within Terminal's command prompt.

For example, to start a new Jekyll site and test it in the browser, you would open Terminal and enter the following commands:

```
$ jekyll new mysite
$ cd mysite
$ jekyll serve
```

(Note: this example is running on a Mac but would work the same once Jekyll is properly installed on Linux, Unix, or Windows.)

Any output resulting from these commands would also be shown in Terminal (see Figure 2-1).

Figure 2-1. The output of starting and serving a new site using Jekyll in the Mac Terminal application.

NOTE

Web-Based Static Site Editors

While most static site development occurs via the command line, there has been recent growth in browser-based editors for static sites.

Prose.io is a free service designed specifically for editing Jekyll projects stored in GitHub. Meanwhile, CloudCannon offers an web-based editor geared towards nontechnical users, allowing them to create and edit pages on a Jekyll site via a web-based site admin. Netlify offers an open source CMS built for building and maintaining static sites.

In addition, the Harp generator also offers the Harp Platform, a commercial service that, while it doesn't offer web-based editing, does allow for easy publishing via Dropbox.

Templating Languages for Theming

A large part of the power of static site development comes in the ability to quickly and easily develop themes. These themes allow developers to customize the look and feel of the site as well as

designate where and how content will be displayed within the final output of the site.

Fortunately, most static site generators rely on pre-existing tools for theming rather than creating their own, proprietary solution. Many static site generators even allow you to choose which solution you use for theming—many doing so via extensions.

For example, Jekyll uses the Liquid Templating language by default. Liquid is an open source templating solution that was originally created for the Shopify ecommerce system. Liquid templates are a mixture of HTML and Liquid markup. For example, the following snippet would loop through the first two posts in a Jekyll blog and output the titles, dates, and excerpts within the HTML shown:

```
{% for post in site.posts limit:2 %}
    <div class="6u">
        <section class="box">
            <a href="{{ post.url | prepend: site.baseurl }}"
class="image featured"><img src="{{ post.banner | prepend:
site.baseurl }}" alt="" /></a>
            <header>
                <h3>{{ post.title }}</h3>
                    <p>Posted {{ post.date | date: "%b %-d,
%Y" }}</p>
            </header>
            <p>{{ post.excerpt }}</p>
            <footer>
                <ul class="actions">
                        <li><a href="{{ post.url | prepend:
site.baseurl }}" class="button icon fa-file-text">Continue
Reading</a></li>
                </ul>
            </footer>
        </section>
    </div>
{% endfor %}
```

Many static site generators built upon Node.js default to the Jade template language, which has a very different syntax. The next example is the same template, but rewritten in Jade for Wintersmith:

```
- var i=0
- var articles = env.helpers.getArticles(contents);
each article in articles
    - i++
    if i<3
        div(class="6u")
            section(class="box")
                a(href= article.url, class="image featured")
```

```
            img(src= article.metadata.banner)
        header
            h3= article.title
                    p= "Posted " + moment.utc(arti
cle.date).format('MMM DD, YYYY')
            | !{ typogr(article.intro).typogrify() }
        footer
          ul(class="actions")
              li
                    a(href= article.url, class="button
icon fa-file-text") Continue Reading
```

There are a multitude of other template language options used by popular static site generators. Each has its pros and cons, which are important to consider whether it comes down to specific features or just stylistic preference.

> **NOTE**
>
> **More Code Samples**
>
> These examples and others throughout this document are taken from my Static Site Samples project on Git-Hub. The project aims to help developers learn the differences between various static site engines by recreating the same example site using multiple engines. As of this writing, there are examples built with Jekyll, Harp, Middleman, Wintersmith, Hugo, and Hexo static site generators.

Local Development Server

Most static site generators include a local server utility that can be spun up within the development directory to allow quick live-previewing of the site. This lets developers easily see their edits and preview content without needing to generate the entire site upon each change (see Figure 2-2).

Most local development servers also include the ability to see changes "live" (i.e., without reloading the browser), similar to LiveReload. This is useful for development using the templating language. However, as many static site generators support pre-processed languages like Sass, LESS, and CoffeeScript, it can dramatically ease the development workflow. Of course, the live preview also previews Markdown content.
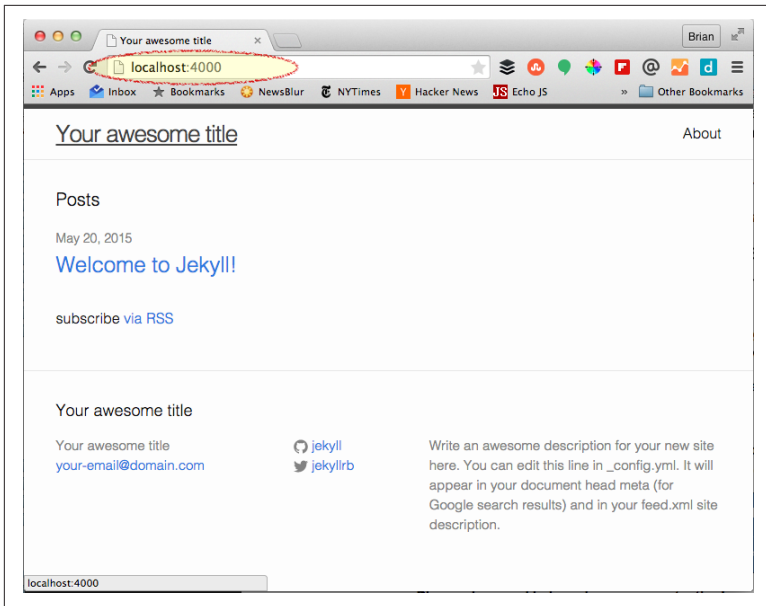
Figure 2-2. Loading the default Jekyll site within Chrome using the local development server.

File-based Data Formats

Almost no site contains purely long-form content, so simply supporting article/post/page content would severely hinder the applicability of a static site generator. Thankfully, most static site generators support one or more file-based data formats like JSON, YAML, and TOML.

File-based data formats are useful for structuring any sort of arbitrary data independent of its display. This allows the designer or developer to both reuse the data in multiple places or change the way it is displayed without duplicating or modifying the original data.

To better understand the value of this feature, let's explore a simple example. I'm a fan of the Cartoon Network show Adventure Time! and I want to create a fan page for the show. I need to display characters on various pages throughout the fan site. First, I'd create a file containing the data (the following example portion uses YAML):

```
-    name: "Finn the Human"
     image: "/images/finn.jpg"
      description: "Finn is a 15-year-old human. He is roughly
five feet tall and is missing several teeth due to his habit
of biting trees and rocks among other things."
-    name: "Jake the Dog"
     image: "/images/jake.jpg"
      description: "Jake can morph into all sorts of fantastic
shapes with his powers, but typically takes the form of an aver
age sized yellow-orange bulldog."
```

Now that I have the character data in a structured format, I can use it to display the characters anywhere on the site. For instance, in the following snippet, I am displaying them on my site's home page using the Middleman generator and the Erb templating language:

```
<section>
    <header class="major">
        <h2>Characters</h2>
    </header>
    <div class="row">
    <% data.characters.each do |character| %>
        <div class="4u">
            <section class="box">
                    <span class="image featured"><img src="<%=
character.image %>" alt="" /></span>
                <header>
                    <h3><%= character.name %></h3>
                </header>
                <p><%= character.description %></p>
            </section>
        </div>
    <% end %>
    </div>
</section>
```

Extensible Architecture

Many, if not most, static site generators support the concept of extensions or plug-ins to either customize the behavior of the generator or add support for additional functionality. Depending on the size of the community for the specific generator, there are often many pre-built plug-ins or extensions available. In most cases, you can build your own, though this requires knowledge of the language upon which the tool was built.

A solid list of community-built extensions is generally a sign of a healthy user community for a particular static site generator. Jekyll, for instance, has a long list available plug-ins as well as instructions

on building your own using Ruby. Middleman, another popular generator built with Ruby, also has an extensive list of available extensions built by its user community. The same goes for Wintersmith, which is built on Node.js.

A Build Process

I saved this for last because it is the defining feature of a static site generator—building static HTML, CSS and JavaScript files.

In most cases, the process is simply a matter of entering the build command on the command line. For example:

```
jekyll build
```

or:

```
wintersmith build
```

Most static site generators offer a variety of options when building, such as verbose error logging or a watch option to continuously rebuild when files change. These are either added as options to the command line, set via some form of configuration file, or some combination of both. In most cases, tacking on `--help` to the command on the command line will give you the available options for building with that generator.

What Skills Are Required for Static Site Development?

The core, common characteristics of static site generators should have helped shed a little bit of light on to what kind of technical skills are required for static site development. Assuming you intend to customize the look and feel or functionality of your site to any degree, static site development will require the same general skill set that any sort of web development requires: web design, HTML, CSS, and JavaScript. Beyond that, though, static site development may require a couple of additional skills.

Comfort Working with the Command Line

Many developers are used to working within IDEs that perform or assist with many of the common day-to-day development tasks. No such tooling currently exists for static site development, so develop-

ers will have to be comfortable using the command line to perform most tasks related to developing, testing and deploying a static site.

Some popular editors do have plug-ins or extensions for working with static site generators, though the scope of their functionality is generally limited, meaning that they will not eliminate the need to work on the command line to perform many tasks. Here are a few examples:

sublime-jekyll
> offers features such as tasks for building new posts/drafts and code completion for Jekyll variables and filters;

middleman-sublime
> simply offers integrated building of Middleman projects within Sublime Text;

Jekyll-Atom
> provides shortcuts for creating new posts, running the Jekyll server and more as well as snippets for common Liquid templating tasks;

brackets-jekyll
> a Brackets extension specifically geared towards building Git-Hub Pages with Jekyll.

It's also important to point out that a majority of debugging generally occurs on the command line. In most cases, errors generated during the testing or building processes will be output to the console.

Ability to Learn and Work with Complementary Languages and Tools

This one sounds confusing, probably because it covers a broad array of tools and languages. The point is that developers working on static sites may encounter one or many of the following within a given project over and above the standard HTML, CSS and JavaScript:

- Markdown
- Sass
- LESS
- Stylus

- CoffeeScript
- Jade
- Liquid
- Erb
- EJS
- Handlebars
- Mustache
- YAML
- JSON
- TOML
- npm
- RubyGems

This is just to name a few! Obviously, you won't be working with all of these at all times, but let's take a typical Jekyll project. By default, it would include:

- Markdown
- Liquid
- YAML
- RubyGems

Depending on the generator and/or the level of customization you require, you may also need some degree of knowledge about the underlying language used to build the tool. For example, if you needed to write an extension for Middleman or Jekyll, you'd need to know Ruby; or for Wintersmith or Hexo, JavaScript (and perhaps an understanding of Node.js). In my own experience, some tools necessitated a knowledge of the underlying language simply due to limited documentation or, as in the case of Hugo, to properly write templates.

Static Site Generators Are Tools for Developers

Just in case this isn't already clear, I want to emphasize that static site generators are built for developers. This starts with the development of the site all the way through to adding content. It's unlikely that nondevelopers will feel comfortable writing in Markdown with YAML or JSON front matter, which is the metadata contained at the beginning of most static site engine content or files. Nor would non-technical users likely feel comfortable editing YAML or JSON data files.

This doesn't mean that you can't use them within a team comprised of developers for site development and writers for content contribution, for example. Still, you'd need to take into account that the developers would likely be more heavily involved in the publishing process than with a traditional database-driven CMS.

What Types of Sites Are Static Site Generators Useful for?

The basic answer to this question is that static site generators are useful for building sites that:

- Focus heavily on delivering content
- Have a low degree of user interactivity
- Update infrequently

It's important to note that all of these criteria are subjective: you may find that a static site is still worthwhile despite not meeting the criteria discussed here. There's no magic formula to determining whether a static site generator is the proper solution for your site. As with most anything in web development, the answer is, "It depends." Nonetheless, let's look at these a little closer at the criteria.

Focus on Delivering Content

While one could make a valid argument that the entire Web is focused on delivering content in some manner, in this case, we're generally talking about informational (and typically textual) content like articles, blog posts, and pages. This is different from, say, a web application that is focused on functionality.

As we've discussed, the internal mechanisms within a static site generator are limited to outputting content from files such as Markdown, HTML, YAML, or JSON, all of which lend themselves to long-form text content (e.g., Markdown, HTML) or short-form text data (e.g., YAML or JSON). Of course, static sites can also support things like video or audio content via the standard HTML `<embed>`, `<video>`, or `<audio>` tags.

Low Degree of User Interactivity

Because the dynamic aspects of any static site must rely on either pre-built or custom-built services, the degree of interactivity in a

static sites tends to be limited. As discussed earlier, there are easy solutions for common interactive elements like comments and calendars, but most static sites don't stray beyond the core functionality these services can provide.

Yes, you can build just about anything using a combination of Ajax and a cloud-based data store. However, if your static site relies upon a lot of complex, asynchronously loaded JavaScript and data, it begs the question as to why it is static site to begin with?

Update Infrequently

Keep in mind that, with each site update, you need to build and push a whole new batch of static files up to the server. For instance, even though you may only have added an article to a subsection of your site, this will likely cause changes to the home page, the navigation, the RSS feed, and so on. This means that a whole new batch of files will need to be compiled and pushed live to support this single change. While there are tools that can ease the build and deploy process (we'll discuss some of these in an upcoming chapter), depending on how frequently your site updates, this could still become an enormous bottleneck to managing your site.

The exact meaning of "infrequently" is open to interpretation and can be impacted by the size and complexity of your site, which can impact the time it takes to build and deploy. If, for instance, you are running a news site, you are likely focused on delivering content and have a low degree of user interactivity but you may add or update content constantly throughout the day to keep up with the news cycle. In this case, deploying as a static site will probably not be appropriate. Nonetheless, it may be a perfect fit for a news magazine that typically publishes articles on a publication schedule, even if that is daily.

Common Use Cases

While static site generators are not built as a one-size-fits-all solution and support all sorts of customizability, there are common types of sites that meet the criteria above and therefore work well as static sites. The most common are:

Blogs

Blogs are the most frequent use case for a static site engine and, in fact, most generally use a blog site as their default site files.

Informational sites or brochureware

A lot of sites around the Web actually fall into this category (even if the term brochureware is often used in a derogatory manner). For instance, sites for restaurants, hotels, tourist attractions, and even many company websites.

Documentation

While sometimes large and complex, documentation sites can also benefit from the file-based structure of a static site, which easily lets them be hosted openly on sites like GitHub, allowing input or pull requests from the user community.

There are plenty of other ways to leverage static site generators, but these three use cases are definitely the most common and obvious. In the next chapter, we'll take a closer look at some of the most popular static site generators to gain a better understanding of which solutions may be better suited to your specific needs.

Popular Static Site Generator Options

At this point, we understand what a static site generator is, that it is a tool geared toward developers and what types of sites it is most useful for. But, with close to 400 options, it can be hard to know where to start. In this chapter, we'll look at a few of the more popular options available to give you a deeper understanding of what differentiates each solution.

In order to give a the broadest overview possible while covering a small subset of tools, I've chosen three options representing different languages: Jekyll, built with Ruby; Wintersmith, built with CoffeeScript (a language that compiles to JavaScript); and Hugo, built with Go.

While covering each engine in depth is definitely beyond the scope of this book, I will take a look at each engine focusing on some key elements:

Getting started
> The ease of setup, cross-platform support, and generation of initial site files

Templating and authoring
> The choice of default template language, the basics of customizing a site, and writing content

Documentation and resources
> A look at the existing documentation and availability of additional community resources

Jekyll

Jekyll was originally created in 2008 by Tom Preston-Werner, founder and former CEO of GitHub. Jekyll is arguably the most popular static site generator currently available—the Jekyll wiki lists over 800 sites built with Jekyll. Part of this popularity is due to the fact that Jekyll is the engine frequently used for running free GitHub Pages.

Jekyll is built on Ruby, though a knowledge of Ruby is really only necessary if you intend to extend the engine itself. Otherwise, although it may rely on Ruby-style conventions, there's no need to have a Ruby background to use Jekyll.
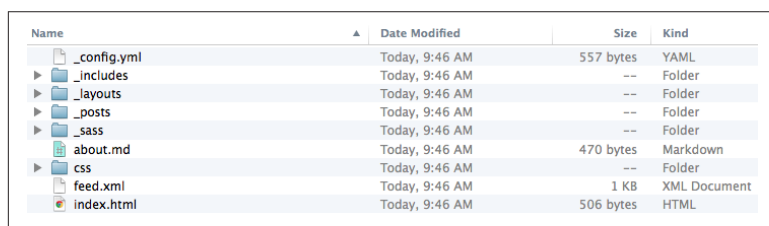
Getting Started

Jekyll is installed via RubyGems. If you're on OS X, installation is simple. Open Terminal and run the following command:

```
sudo gem install jekyll
```

Jekyll is not officially supported on Windows. The Jekyll documentation offers a link to a walk-through covering how to get Jekyll running on a Windows machine. While not as simple as the one command install on OS X, I have personally had success using these instructions on a Surface Pro running Windows 10.

Once you have successfully installed Jekyll, generate the default site files with the following command, where [project name] is the folder you want Jekyll to create for your site (see Figure 3-1):

```
jekyll new [project name]
```

Name	Date Modified	Size	Kind
_config.yml	Today, 9:46 AM	557 bytes	YAML
▶ _includes	Today, 9:46 AM	--	Folder
▶ _layouts	Today, 9:46 AM	--	Folder
▶ _posts	Today, 9:46 AM	--	Folder
▶ _sass	Today, 9:46 AM	--	Folder
about.md	Today, 9:46 AM	470 bytes	Markdown
▶ css	Today, 9:46 AM	--	Folder
feed.xml	Today, 9:46 AM	1 KB	XML Document
index.html	Today, 9:46 AM	506 bytes	HTML

Figure 3-1. The default generated Jekyll files.

Let's do a quick overview of the files and folders that Jekyll generated for you:

_config.yml

The YAML configuration file for your site.

_includes

Where any template partials should be placed (we'll discuss what these are later in this chapter).

_layouts

Holds templates for posts. The template a post uses can be configured in its "front matter" (we'll discuss what front matter is later on).

_posts

Holds the Markdown files for your blog posts.

_sass

Contains Sass includes. Sass is a CSS preprocessor. It is supported by default in Jekyll, but it is not required that you use it. Plain CSS works fine if you prefer. If you do not use Sass in your site, this folder is not required.

css

Contains your site's CSS or primary Sass files.

feed.xml

A template that generates an RSS feed for your site.

index.html

The template for your site's home page.

You can find out more about the directory structure of a Jekyll site in the documentation, including optional folders that Jekyll did not autogenerate for you.

If you explore the generated files, it is worth noting that any file or folder name that starts with an underscore will not have a corresponding file when the site files are generated. For instance, in Figure 3-1 you can see the default files: *about.md* will generate an about.html file in the generated site, but *_config.yml* will not have a corresponding file in the generated site.

To test the newly created project using Jekyll's local server, first change directory into the project folder and start Jekyll's local server:

```
cd [project name]
jekyll serve
```

There are a number of configuration options available for the Jekyll server, but by default, it will run the site on port 4000, meaning the running site will be accessible in a browser via *http://localhost:4000*. For the full list of server options, use the `jekyll serve -h` command.

Templating and Authoring

As discussed, Jekyll uses the Liquid template library by default, though it does support additional templating options via extensions. Let's look at some of the basics of building templates.

Template basics

Outputting the contents of variables within a template requires wrapping the variable in curly braces:

```
<h2>{{ page.title }}</h2>
```

In this case, `page.title` is a default variable that Jekyll makes available on all pages. The documentation contains a full list of default variables that Jekyll provides. You can have custom site and page (i.e., post) variables as well. These are configured in the *_config.yml*, for site variables, or in the post's front matter, for page variables.

Liquid also includes a number of filters that allow for formatting of output. A full list of standard filters can be found in the Liquid documentation, and the Jekyll documentation has details on the additional filters that Jekyll provides. For example, the following standard filter formats the output of dates:

```
<p>Posted {{ post.date | date: "%b %-d, %Y" }}</p>
```

Liquid allows you to separate and organize your templates into different files using partials, which are essentially includes. Includes are placed in the *_includes* folder on the root of a Jekyll site. For example, the following line would include a file named *header.html* within the *_includes* folder:

```
{% include header.html %}
```

Liquid templates can contain conditionals. In the following code, a portion of the template is only shown if the user is on the site's home page:

```
{% if page.url == "/index.html" %}
    <section id="banner">
        <header>
            <h2>Clever Title</h2>
        </header>
    </section>
{% endif %}
```

It's also possible to loop through a collection, such as an array of articles or blog posts:

```
{% for post in site.posts %}
    <div>
        <section>
            <header>
                <h3>{{ post.title }}</h3>
                    <p>Posted {{ post.date | date: "%b %-d,
%Y" }}</p>
            </header>
            <p>{{ post.excerpt }}</p>
            <footer>
                <ul>
                        <li><a href="{{ post.url | prepend:
site.baseurl }}">Continue Reading</a></li>
                </ul>
            </footer>
        </section>
    </div>
{% endfor %}
```

Loops allow things like limits and offsets to further specify the output of a loop.

There is a lot more to templating in Jekyll that we can cover here. Refer to either the Liquid or Jekyll documentation for more details.

Content authoring

By default, Jekyll pages or posts are authored in either HTML or Markdown. The key things to understand about authoring for Jekyll are the concept of Front Matter and the importance of naming.

Front Matter. Front matter is YAML metadata that can be included at the beginning of any file within Jekyll. This includes Markdown and HTML content, as well as CSS or JavaScript files. However, every post that is placed in the *_posts* folder generally includes the following front matter:

Layout

Specifies which layout file (from the *_layouts* folder) will be used when generating this post.

Categories

A space-separated list that defines a subfolder in which a post will be placed when the site is generated. For example, a post with `categories: programming javascript` will be generated into a */programming/javascript/* subdirectory. If a post only has one category, the singular `category` property can be used.

Date

This is the only predefined variable specific to posts and overrides the date that is parsed from the post's filename (more on that in a moment). This can be useful for ensuring that posts are properly date-sorted by Jekyll.

Title

Title is not technically a predefined variable according to the documentation, but posts generally include a title property (in fact, the default generated posts do).

Using just the default values, a post's front matter might look like this:

```
---
layout: post
title:  "The Law Offices of Gabriel John Utterson"
date:   1886-01-05 09:00:00
categories: services
---
```

It's also worth noting that you can add any arbitrary metadata to a post's front matter and access it via the `page` variable. For example, if I were to add a description to the front matter, it could be accessed as `page.description`.

Naming. Jekyll expects posts to be named in a particular manner. Posts must be placed in the *_posts folder* and must be named using the format of *year-month-day-title.markdown* (or *.md*). Year should be a four-digit number, while month and day should both be two digits.

As an example, a post published on June 5, 2015 and named "Hello World" should be named *2015-06-05-hello-world.markdown*. The title portion of the filename doesn't have to match the actual title of

the page in the metadata. When the final page is generated, the URL will have the format */2015/06/05/hello-world.html*. This portion of the URL follows any category specified in the metadata. Thus, by default, if our "Hello World" post was in the category "general," the full URL would be */general/2015/06/05/hello-world.html*. This URL can be configured if you would prefer a different URL format.

Documentation and Resources

The documentation for Jekyll is reasonably comprehensive, but there are ample additional resources available. This is not a comprehensive list, but the following are some resources that go beyond the information in the official documentation:

Getting started
- "Getting Started with Jekyll" by Brian Rinaldi (yes, that's me!)
- "Building Static Sites with Jekyll" by Andrew Burgess
- "Learning Jekyll by Example" by Andrew Munsell (requires purchase)
- Jekyll Now, a tool for creating a Jekyll blog without touching the command line, by Barry Clark

Themes and plug-ins
- Jekyll Themes by Matt Harzewski
- Jekyll-Plugins by Grok Interactive
- Poole, the Jekyll "butler," by Mark Otto

Migration
- Wordpress-to-Jekyll Exporter by Ben Balter

Wintersmith

Wintersmith is one of the more popular Node-based static site generators. It was created by Johan Nordberg in 2013.

One of the differentiators of Wintersmith is that it tries to be less prescriptive about how you structure your site or name your files. It also aims to make it easy to extend the capabilities of the generator through the enormous number of resources available in npm, the package manager for Node.js.

In my opinion, working in Wintersmith will require a reasonable comfort level with JavaScript programming, as JavaScript is often necessary for writing Wintersmith templates. However, Wintersmith was actually written in CoffeeScript. It's not entirely necessary that you understand CoffeeScript to use Wintersmith, but the limited documentation often necessitates referencing the source code of the project. Thus, I believe the ability to read and understand Coffee-Script code is definitely helpful.

Getting Started

Wintersmith is installed via npm, which comes with Node. If you don't already have Node installed, you'll want to download and install that first.
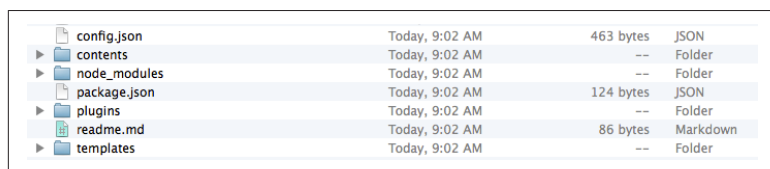
Once you have Node and npm installed, installing Wintersmith is a single command on Windows or OS X. Open Terminal or the command prompt and enter the following command:

```
npm install -g wintersmith
```

It should be noted that on OS X you may need to use the `sudo` command to perform a global install (the `-g` indicates a global install for npm).

To create a new site with Wintersmith, use the `new` command (where `[project name]` is the folder you want Wintersmith to create the new site files in); see Figure 3-2:

```
wintersmith new [project name]
```


config.json	Today, 9:02 AM	463 bytes	JSON
contents	Today, 9:02 AM	--	Folder
node_modules	Today, 9:02 AM	--	Folder
package.json	Today, 9:02 AM	124 bytes	JSON
plugins	Today, 9:02 AM	--	Folder
readme.md	Today, 9:02 AM	86 bytes	Markdown
templates	Today, 9:02 AM	--	Folder

Figure 3-2. The default Wintersmith generated files.

The root Wintersmith files all relate to configuration and customization of the site. The site's content is contained within the contents folder, while the look and feel of the site is determined by the files in the templates folder.

To test the generated site, change directory into the project folder and start Wintersmith's local preview server:

```
cd [project name]
wintersmith preview
```

By default, the server runs on port 8080, so you can access it at the URL *http://localhost:8080*. It is also verbose, meaning that it will output detailed error messages and loaded resources to the console. These features and others are configurable via options. To view the preview options, use `wintersmith preview -help` command.

Templating and Authoring

As is common with Node.js-based static site generators, Wintersmith defaults to Jade for templating, though it does support a number of other template engines via its available plug-ins. Jade touts its terse syntax, which is very different from standard HTML. For instance, it has no brackets, or closing tags, and indentation matters. Let's dive into some of the basics.

Template basics

There are a number of ways to output variables in Jade. To output the value of a variable within a code block, use an equal sign. The following code would output the value of the `page.title` variable within an h2 block:

```
h2= page.title
```

There is some information about what variables a Wintersmith page makes available in the documentation.

One way to concatenate a variable with other string contents within a tag block is to use the following syntax:

```
h2= The title is #{page.title}
```

Variables can also be output using an equal sign within attributes on a tag, which are placed within parentheses (Note: the `article` variable in the following example is not a default variable.):

```
a(href= article.url, class="image featured")
```

It's worth noting that by default, variables are escaped, meaning that special characters are turned into HTML entities. Jade allows you to specify that you would like the contents of variables to remain unescaped by using an `!`. More details on how this all works are available in the Jade documentation.

Jade doesn't include any formatting helpers by default. However, Wintersmith can be extended with npm modules, many of which offer functions for formatting or filtering output. Wintersmith includes several of these by default: Moment.js, Underscore.js, and Typogr. Let's look quickly at how you would use Moment.js to format a date within a Wintersmith template:

```
p= "Posted " + moment.utc(article.date).format('MMM DD, YYYY')
```

You can split templates into smaller chunks to make them more manageable and reusable using includes. The following code includes a *header.jade* file from a partials folder that is within the templates directory:

```
include ./partials/head
```

You can also separate portions of templates using template inheritance.

Jade allows for conditionals. The following snippet only shows if the user is on the site's home page:

```
if !page
    section(id="banner")
        header
            h2 Clever Title
```

It's also possible to loop through arrays of content, such as posts or data. The following code loops through the full list of posts:

```
- var articles = env.helpers.getArticles(contents);
each article in articles
    div
        section
            header
                h3= article.title
                    p= "Posted " + moment.utc(article.date).for
mat('MMM DD, YYYY')
                | !{ typogr(article.intro).typogrify() }
                footer
                    ul
                        li
                            a(href= article.url) Continue Reading
```

There are two important thing to note about the preceding code. First is the way the `articles` variable is defined. Using the `-` prefix on a line allows you to include any arbitrary JavaScript that is evaluated during the compile process. In this case, we are defining a variable named `articles` that can then be used in the template. This

line will not exist in the compiled HTML page. Second, the pipe (i.e., |) before `!{ typogr(article.intro).typogrify() }` indicates that this line is plain text, meaning it isn't wrapped in a tag.

This has just been a basic overview of what is possible in Wintersmith templates. For a more detailed look, check the Jade or Wintersmith documentation.

Content authoring

Posts in Wintersmith are written in Markdown. It uses the marked renderer for Markdown, although other renderers are available as plug-ins. By default, posts are placed in the contents/articles folder.

One major difference between Wintersmith and Jekyll is that each post is typically given its own directory, which will determine its SEO-friendly URL (although this is not a requirement). For instance, a "Hello World" post would be placed in a directory named *articles/hello-world*. Inside that directory would be a Markdown file for the post named *index.md*. There is no specific file-naming format required.

Each Markdown post can have metadata on top (aka front matter) using the YAML format. No metadata is required, but template, title, and date are commonly specified:

```
---
title:   "Hello World"
date:    2015-06-08 10:33:56
template: article.jade
---
```

The `template` property designates the template that will be used to render the post, `title` is the title of the post, and `date` is the date it was posted. It's important to note that if template isn't specified, the post will not be rendered (which likely isn't what you intended). If `title` and `date` are unspecified, their values will receive defaults.

Wintersmith allows you to specify any arbitrary metadata within the front matter. It is accessible within a template via a `metadata` property on the `page` object.

You can also pass metadata to a template using JSON rather than YAML front matter. For more details on that, refer to the documentation.

Documentation and Resources

The documentation for Wintersmith is very limited, consisting mainly of a quick-start guide. Because Wintersmith can also be used programmatically within Node.js, there is an API guide, but it offers little assistance when developing a site using the standard command-line generator. Here are some resources that can supplement the documentation:

Getting started
- "Getting Started with Wintersmith: A Node.js-based Static Site Generator" by Brian Rinaldi (yup, that's me, again)
- "Creating Posts, Custom Metadata, and Data in Wintersmith" by Brian Rinaldi
- "Introduction to the Wintersmith Process" by David Tucker, rundown of getting set up but looks at using Nunjucks for templating within Wintersmith

Themes and plug-ins
- Wintersmith Showcase: many of the projects listed are open source, so you can borrow techniques by viewing the source code.

Hugo

There are two key things that make Hugo different than most other static site generator options. The first is that it is one of only a handful of generators written in the Go programming language, a language originally developed by Google. The second is that it focuses on extremely fast build times. This can be an important consideration, as build times using other engines can become a significant impediment as the size of a site grows.

As we'll discuss in more detail in the next section, Hugo templates are built using the Go template language. In my own experience, if you are unfamiliar with the Go language, the syntax can be a significant departure from what you are used to. Also, some aspects of building even basic templates require at least a basic knowledge of the Go language, though this can be gleaned from the Go language documentation.

Getting Started

Installing Hugo is mostly a matter of downloading the proper binary executable for your platform from the releases page. Hugo supports Windows, OS X, Linux, and FreeBSD. I say mostly because you will likely want Hugo on your PATH to make it easy to access from anywhere via the command line.

On Windows, this requires accessing the Environment Variables within the Advanced System Settings in your Control Panel. On OS X, the easiest way to install Hugo is using Homebrew, which will automatically take care of making it available via the $PATH variable. If you have Homebrew installed, you can simply enter the following command via Terminal:

```
brew install hugo
```

Once Hugo is installed, it's time to generate the files to begin a new site. Similar to the other two engines, creating the default files (see Figure 3-3) uses the new command as follows (where [project name] is the folder you want Hugo to place the new site files in):

```
hugo new site [project name]
```

Hugo offers the option of specifying the kind of data format you would like to use for a site when generating new site files. By default, Hugo uses TOML, but you can add add the option --format="yaml" when generating your site if you prefer to use YAML, as the other engines we discussed have used.

Figure 3-3. The default site files generated by Hugo. All of the folders are empty to start.

The base site files and folders are all empty except for the configuration file. In the next section, we'll look at how you can install a theme and generate content.

Of course, Hugo also includes a local web server to test your site. The site is empty right now, but once you've created some content, enter the following command from the command prompt:

```
hugo server
```

The default port is 1313, so to open the site in a browser, you would use the URL *http://localhost:1313/*. There are a number of options for the server, such as specifying the port or watching for file changes. To get a full list of options, use `hugo server --help`.

Templating and Authoring

Hugo doesn't create any layout files or install a theme of any sort by default. You have two options to start, either creating layouts in the */layouts* folder or installing a theme. There is a long list of available themes and instructions on how to install them via Git.

Building your own layouts requires using the Go html/template library. The official documentation on this topic isn't a friendly read, although the Hugo documentation offers a good primer on the topic. Let's look at some of the basics.

Template basics

To output variables within a template, you surround them by curly braces:

```
<h2>{{ .Title }}</h2>
```

Hugo has a long list of variables that can be used in templates depending on the context. For instance, the `.Title` variable in the preceding snippet is a page variable. The dot in the example refers to the current context, which, in this case, is a page.

To format the output of variables, you can use the available utilities within the Go language. For example, to format the output of the date, we can use Go's date/time format function:

```
<p>Posted {{ .Date.Format "Jan 2,2006" }}</p>
```

The Go by Example site offers other examples of string formatting using the Go language.

Hugo templates can be split into separate files to make them easier to maintain and reuse. The following code includes a *header.html* file from a partials subfolder that is within the layouts directory:

```
{{ partial "header.html" . }}
```

Notice that the partials folder is left off the path. You can organize partials within subfolders, but the partials folder should still be left off of the path.

Go templates let you use conditionals within your layouts. In the following code, the banner is only shown if the user is on the home page of the site rather than a post or page:

```
{{ if .IsNode }}
    <section id="banner">
        <header>
            <h2>Clever Title</h2>
        </header>
    </section>
{{ end }}
```

The `.IsNode` variable is a page variable that is always false on a page.

Loops use a `range` construct that functions somewhat differently from the Liquid or Jade samples earlier in the chapter. For example, the following snippet loops through all posts:

```
{{ range .Site.Pages }}
    <div>
        <section>
            <header>
                <h3>{{ .Title }}</h3>
                <p>Posted {{ .Date.Format "Jan 2,2006" }}</p>
            </header>
            <p>{{ .Summary }}</p>
            <footer>
                <ul>
                        <li><a href="{{ .Permalink }}">Continue
Reading</a></li>
                </ul>
            </footer>
        </section>
    </div>
{{ end }}
```

It is also possible to use the first keyword to limit the loop, such as in the following code example, where we only output the first five posts:

```
{{ range first 5 .Site.Pages }}
```

If you need to access the index of a range item, however, you'll need to define the range somewhat differently:

```
{{ range $index, $element := .Site.Pages }}
```

Now I can access the $index for a zero-based index of the item in the range.

The Hugo documentation on templates does a good job of covering the basic information you'll need to continue developing templates. In my experience, you will, nonetheless, need to keep a tab open to the Go language documentation as well.

Content authoring

One of the nice things about Hugo is that it offers a command to generate new content, which can be helpful for at least laying out the front matter metadata format for a new post.

To create a new page or post, open up Terminal or the command prompt and type the following command to create a new "About Us" page:

```
hugo new about-us.md
```

> **Hugo Has No Default Templates**
>
> It's important to note at this point that Hugo does not generate default templates for your site. If you are working off of the initially generated set of default site files, but have not yet installed a theme or created layout files, you will still be unable to preview your "About Us" page. You can either follow the instructions for installing a theme or create the necessary layout files yourself.
>
> To create these yourself, you'll need, at a minimum, a home page and a single post template within the layouts folder. The home page template is named *index.html*, and the default single post page would be placed in *layouts/_default/single.html*. For a good, minimalist example of a Hugo site, check out my Static Site Samples project on GitHub.

Hugo will generate a basic Markdown file with the default front matter metadata in the default format, TOML. For the most part, TOML is very similar to YAML, but the formatting is slightly different. The following code snippet is the default TOML front matter for our "About Us" page:

```
+++
date = "2015-06-10T14:42:51-04:00"
draft = true
title = "about us"

+++
```

If you prefer YAML for front matter, you can use the command `new about-us.md --format="YAML"` to create the "About Us" page. The preceding `draft` value tells Hugo not to render this page yet, as it is still being written.

According to the documentation, Hugo requires the following values for front matter: title, description, date, and taxonomies (i.e., tags or categories). The latter two values are not included in the default generated content file for some reason. Fortunately, Hugo supports the concept of archetypes, allowing us to specify additional default metadata we want in the file.

In the archetypes folder under the site root, create a *default.md* file. Within that file, we'll put some default content:
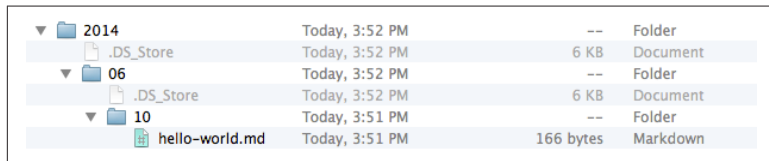
```
+++
description = "My amazing new post"
tags = [
  "x",
  "y"
]
categories = [
  "x",
  "y"
]
+++

#My Amazing New Post
```

After saving this file, when you enter the `new about-us.md` command, it will not only have the title and date properties, but also the description, tags, categories, and Markdown title as specified in the *default.md* file. This can be helpful to create boilerplate for writing new content.

You can also specify any arbitrary front matter metadata you choose. This can be accessed via the `.Params` variable on a page. For example, if you were to add a banner value in the front matter to specify a banner image, it could be accessed via `.Params.banner`.

The last thing to note is that, although Hugo puts new posts in the content folder by default, you can organize the content within that folder however you choose. If you wanted the SEO URL for your post to be *2014/06/10/hello-world.html*, then your folder structure within the content folder would look like Figure 3-4.

Figure 3-4. A Hello World post with the URL of /2014/06/10/hello-world.html.

Hugo offers a lot of customization options for creating content. The documentation does a generally good job of covering all of the details about creating content.

Documentation and Resources

The good news about Hugo is that the documentation is pretty thorough. As I've mentioned a couple times already, you will need to occasionally refer to the Go language documentation to supplement the information.

The bad news is that, as search terms go, Hugo and Go are both awful, making finding additional resources an often difficult matter. Here are a few that are worth exploring:

- "Migrating to Hugo from Octopress" by Nathan Leclaire, Octopress is a Jekyll-based static site generator.
- (Hu)go Template Primer by Daisuke Tsuji.
- "Build Static Sites in Seconds with Hugo" by Dan Hersam. Note that this is a commercial course through Udemy. I have not personally reviewed the content, but the curriculum offers two hours of material covering everything from setup to deployment of a Hugo site.

And So Many More...

In this chapter, we looked at only a few of the approximately 395 static site generators currently available. The three we looked at were chosen not just because each represents a different underlying language—Ruby, JavaScript/CoffeeScript, Go—but because they each have very different approaches to building static sites.

Obviously it is impossible to try even a small fraction of the available static site generators before making a decision on what to choose for your project. While the right option depends heavily on the requirements for your project, my best advice would be to use the following criteria when evaluating:

What language is it built in?
　　While it isn't a requirement that you know the language that the tool was built with, it can certainly help in some important ways. For instance, it opens up the possibility of extending the core through extensions or even by modifying the source (which you could even contribute back to the project). It can be useful for debugging some difficult errors. Lastly, it is a good backup when the documentation and available resources fall short—you can read the source code.

How recently was it updated?

Open source projects get abandoned. It's worth checking the repository for the project and ensuring that there has been some activity within the last six months to a year. If there hasn't, this could be a sign that the project is dormant or dead.

How thorough is the documentation?

Many, if not most, static site generators suffer from the same problem that afflicts many open source projects: a lack of useful documentation. Be sure you spend time looking through the documentation, as even some that look complete at first glance are often lacking in critical details or examples.

How many third-party resources are available?

Unless the project is brand new, you should be able to find people posting guides and resources for how to use the tool and/or answering questions on sites like Stack Overflow. If you don't, this could be a sign that adoption of the tool has been limited. This can mean that you won't be able to easily get help with issues and generally increases the chances that the project will eventually be abandoned.

In my opinion, unless you have specific reasons for choosing otherwise, such as you are uncomfortable with Ruby and want a tool built in a language you already know, the safest option is Jekyll. It is consistently maintained, has good documentation, and there are a wide array of third-party resources and tools that support it.

Deploying a Static Site

Once you've built your static site using the generator of your choice, it's time to share it with the world. The good news is that, because these are static files, there's no complicated database deployment or environment to set up. Deploying a static site is simple.

This simplicity also opens up a huge number of options. In this chapter, we'll look at just a few of the deployment and hosting options available to you for your static site.

FTP

Since you are dealing with static assets—typically just HTML, CSS, JavaScript, and images—deploying to just about any host via FTP requires no special setup. The only important thing to remember is that you need to build your site first; you do not upload the templates and Markdown.

Building Your Site

Some static site generators, like Jekyll, generate the site files whenever you preview the site. Nonetheless, it is best practice to do a build of the site before deploying. All of the static site generators have a build command of some form, often with some options. Let's look at the process using the three generators we covered in the prior chapter.

In Jekyll, you'd use:

```
jekyll build
```

You can have Jekyll rebuild the site every time you make a change by adding `-w` to enable the watch option. For a full list of Jekyll build options, add `--help`.

In Wintersmith, you'd use:

```
wintersmith build
```

If you want to force Wintersmith to clear the output folder before building, use the `-X` option. For a full list of Wintersmith build options, add `--help`.

In Hugo, you'd use:

```
hugo
```

Hugo's build process also includes the option to watch for changes with the `-w` option. For a full list of Hugo build options, add `--help`.

Assuming there were no errors, once you run the build process, your static files will be in the output directory that you set in your site configuration. But what happens if you do encounter an error?

Debugging
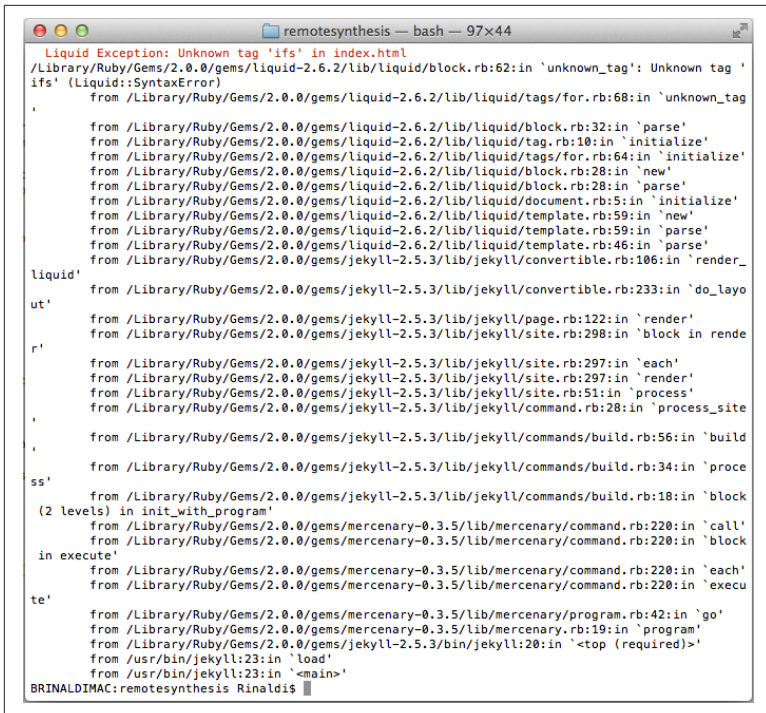
A majority of the debugging experience with static site generators is done via the console log within the terminal. This experience can differ with each generator and often depends on the underlying language or tools upon which the generator was built.

Some generators enable verbose logging in the console by default, and others do not. Verbose logging can be useful when trying to debug complex issues, but often it is extraneous information (see Figure 4-1).

Some generators will also give you error details when previewing a site in the browser; others fail silently in the browser but display errors in the console (see Figure 4-2).

The point is, there is no consistent debugging experience across static site generators, and the current experience often leaves much to be desired.

Figure 4-1. A Jekyll build error with verbose logging enabled. Jekyll does not provide verbose logging by default. To enable it, use the -v option when running a build.
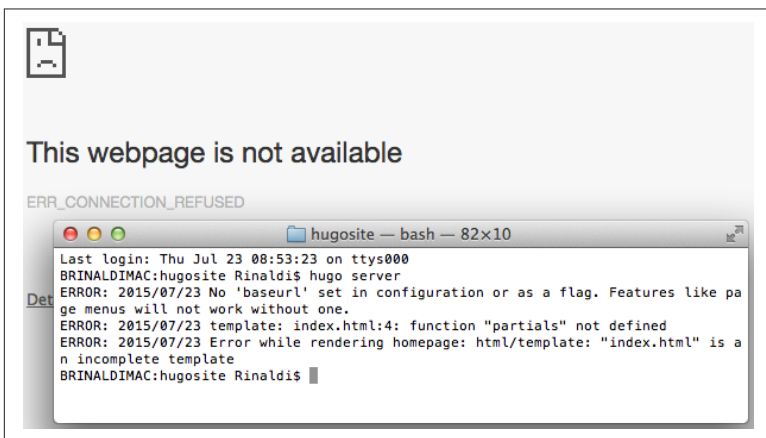

Figure 4-2. Hugo displays error messages in the console but simply fails to display at all in the browser.

Once your site is built, open your FTP client, for example Filezilla or Transmit, and push the files to your host. One thing to remember is that some static site generators regenerate the entire site upon each build, so the sync feature available on many FTP clients will simply push the entire site, regardless.

> **NOTE**
>
> **Glynn**
>
> Glynn is a tool that combines the build and deploy via FTP steps into a single command or Jekyll-based sites. To set up Glynn, you first need to install it via Ruby Gems:
>
> ```
> gem install glynn --source http://gemcutter.org
> ```
>
> Then configure it via your site's _config.yml. There are a number of configuration options, but setting the host, root directory, and passive mode are the required options. In the following example configuration, username is included, meaning it will not have to be re-entered upon each use:
>
> ```
> #glynn
> ftp_host: 'your_site.com'
> ftp_dir: 'site/root'
> ftp_passive: true
> ftp_username: 'your_username'
> ```
>
> Once properly configured, enter the command `glynn` from the command line/terminal. Glynn will ask for your FTP password and, once entered, will push the files live.

GitHub Pages

GitHub and its free website hosting service, GitHub Pages, probably had a lot to do with the popularity of Jekyll. While technically GitHub Pages can host any static site, not necessarily ones created with Jekyll, there is close integration with Jekyll that makes it the de facto option. Let's look at the how it works, since, in my opinion, the official documentation on Jekyll's site makes it seem more complicated than it really is.

There are two types of GitHub pages: user/organization pages and project pages. It's important to note that although we call these "pages," they are actually full sites, not a single page. Let's see how to create a user page.

First, go to GitHub and create a new repository. This repository must be named *[username].github.io*. For example, my username on GitHub is remotesynth, so my repository is named *remotesynth.github.io*. Next you'll need to clone the new repository, either using the command-line Git client or the GitHub desktop app.

At this point, your repository is empty. Via the command line, change directory into the new repository and enter `jekyll new .`, which will create a new Jekyll site in the current directory. Open the *_config.yml* and modify the configuration with your correct site details.

Finally, just check the source of your Jekyll site into the GitHub repository. There's no need to run a build (and the *.gitignore* file is set up to ignore the *_site* directory by default anyway, so don't worry if you ran a preview). Your new GitHub Pages user site should already be up and running at *[username].github.io*, albeit using the default Jekyll blog files.

GitHub Pages is an easy and free solution for sites such as a personal blog or a project blog. You can even set up a custom URL as opposed to using a *github.io* subdomain. GitHub offers additional documentation covering how to set everything up and run a Jekyll blog on GitHub Pages.

> **NOTE**
>
> **Surge**
>
> Another hosting option for static sites similar to GitHub Pages is Surge. Surge offers free publishing and custom domains on its basic account and more advanced options via a commercial offering. The benefit of Surge is that it makes the deployment process transparent via an easy-to-use command-line tool. It also offers integration options for various other build tools so that it can easily fit within an existing build and deployment process. Surge includes instructions for deploying Jekyll sites, but it will work with any static site generator.

Cloud Hosting

The list of cloud hosting options seems to grow every day, but any cloud host should easily be able to handle a static assets. Many of the popular static site generators include plug-ins, third-party tools, or

direct integration for deploying to some of the more widely-used cloud hosting services like Amazon EC2, Heroku, Azure, and Modulus.

With the growth in popularity of static site generators though, some services have cropped up that offer cloud hosting services specifically targeting these tools. They offer some added conveniences that a traditional hosting service cannot because of their integration with the generators. Let's look at a couple of options.

Netlify

Netlify is a commercial service that offers a cloud hosting solution specifically designed for static site generators. You can use their command-line tool to push static assets live. Even better, connect it to a GitHub or Bitbucket account and have it continuously build and deploy your application whenever new or updated files have been checked in.

The benefit of the latter option is that the build process occurs entirely on the Netlify servers. This means that there is no need to handle deployment of the static assets at all. Netlify pulls changes from the repository and updates the live site with the results of running the build command, which is configurable.

For example, I was able to get my personal blog, built with Jekyll, building and deploying on Netlify with only the addition of a Gemfile and Gemfile.lock. The gemfile only defined my Jekyll dependency:

```
source 'https://rubygems.org'
gem 'jekyll'
```

The Gemfile.lock will be automatically generated when you run `bundle install` after the Gemfile is created.

Once the files are committed to the repository, Netlify will automatically try to build the site and, if successful, it will immediately be available live.
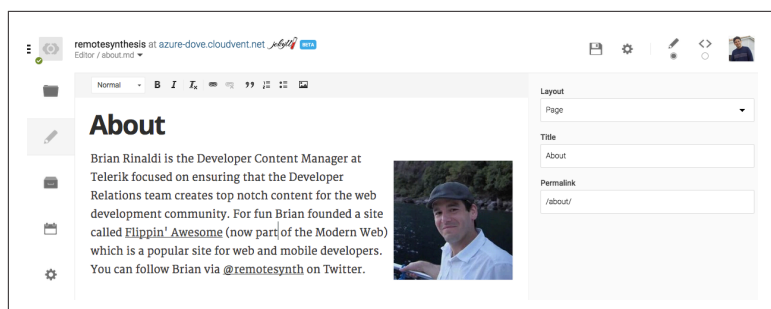
CloudCannon

CloudCannon is a commercial cloud hosting service designed specifically for static sites built with Jekyll. The key difference between CloudCannon and the alternatives is that it offers a web-

based user interface designed to allow nontechnical users to add or edit content on the site.

CloudCannon also integrates with repositories on GitHub or Bitbucket or even a specialized folder on Dropbox. This means that it will automatically sync and build changes to your site whenever they are checked in, eliminating the need for the build-and-deploy step.

Once the files are synced, they can be edited via the browser-based interface using a WYSIWYG-style editor for Markdown. For instance, Figure 4-3 is a screenshot of the editor that allows me to make modifications to my "About" page on my personal blog.

Figure 4-3. Modifying my "About" page using CloudCannon's WYSI-WYG editor for Markdown pages.

As you can see in the screenshot, CloudCannon even allows me to edit the Jekyll front matter using a form-based interface (on the right-hand side of the screen). There's also a web-based editor for Jekyll collections.

The Possibilities Are (Almost) Endless

We've only managed to cover a handful of the many deployment possibilities available to you for your static site. Because we are dealing in purely static files, there aren't the typical deployment limitations that may come with hosting a dynamic server-side language with a database server. A static site can be hosted just about anywhere, and usually extremely cheaply.

As I hope this and the previous chapters illustrate, however, is that *static does not mean simple*. Static site generators offer a lot of flexibility and power. You can build anything from a simple blog to a complex documentation site to a business website.

Yes, some of these tools can often seem obtuse and hard to use at times, especially depending on your level of comfort with the command line and Markdown, but, if you are able to overcome some of the initial hurdles in development, a static site can be remarkably easy to maintain, with the added benefits of speed, security, and simplicity.

About the Author

Brian Rinaldi is the content and community manager on the developer relations team at Telerik, where he is responsible for running the Telerik Developer Network site. He has over 15 years of web development experience and in recent years has focused on front-end web development and mobile development using web technologies. Brian speaks frequently at conferences such as Fluent, DevNexus, and the inaugural JekyllConf and has served on the conference committees for Fluent and QCon NY. Brian writes frequently for publications including the Telerik Developer Network, SD Times, SitePoint, and InfoQ. He is also coeditor of the Mobile Web Weekly newsletter. You can follow Brian on his blog at RemoteSynthesis.com or on Twitter as @remotesynth.