



Jekyll theme for documentation — mydoc product

version 6.0

Last generated: October 28, 2021



© 2021 Blain's World. This is a boilerplate copyright statement... All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Table of Contents

Overview

Get started	3
Introduction	4
Supported features	5
About the theme author	10
Support.....	11

Release Notes

6.0 Release notes	12
5.0 Release notes	14

Installation

About Ruby, Gems, Bundler, etc.	16
Install Jekyll on Mac	24
Install Jekyll on Windows	29

Authoring

Pages	32
Posts	39
Lists	41
Conditional logic.....	45
Content reuse.....	50
Collections.....	52
WebStorm editor tips	54
Atom editor tips.....	58

Navigation

Sidebar navigation.....	59
YAML tutorial in the context of Jekyll.....	62
Tags.....	73
Series.....	79

Formatting

Tooltips.....	82
Alerts	83

Icons.....	91
Images.....	98
Code samples	103
Labels	104
Links	105
Navtabs	106
Tables.....	110
Syntax highlighting	114
Workflow maps	117

Handling reviews

Commenting on files	121
---------------------------	-----

Publishing

Build arguments	124
Themes.....	127
Generating PDFs	128
Help APIs and UI tooltips	141
Search configuration	153
iTerm profiles.....	157
Pushing builds to server	159
Publishing on Github Pages.....	160

Special layouts

Knowledge-base layout.....	163
Glossary layout.....	166
FAQ layout.....	169
Shuffle layout.....	170

Troubleshooting

Troubleshooting	173
-----------------------	-----

Blain's World @ ACC

Welcome to my GitHub Pages



Introduction

Overview

This site provides documentation, training, and other notes for the Jekyll Documentation theme. There's a lot of information about how to do a variety of things here, and it's not all unique to this theme. But by and large, understanding how to do things in Jekyll depends on how your theme is coded. As a result, these additional details are provided.

The instructions here are geared towards technical writers working on documentation. You may have a team of one or more technical writers working on documentation for multiple projects. You can use this same theme to author all of your documentation for each of your products. The theme is built to accommodate documentation for multiple products on the same site.

Survey of features

Some of the more prominent features of this theme include the following:

- Bootstrap framework
- [Navgoco multi-level sidebar](#) for table of contents
- Ability to specify different sidebars for different products
- Top navigation bar with drop-down menus
- Notes, tips, and warning information notes
- Tags for alternative navigation
- Advanced landing page layouts from the [Modern Business theme](#) .

Getting started

To get started, see [Getting Started \(page 3\)](#).

Supported features

Summary: If you're not sure whether Jekyll and this theme will support your requirements, this list provides a semi-comprehensive overview of available features.

Before you get into exploring Jekyll as a potential platform for help content, you may be wondering if it supports some basic features needed to fulfill your tech doc requirements. The following table shows what is supported in Jekyll and this theme.

Supported features

Features	Supported	Notes
Content re-use	Yes	Supports re-use through Liquid. You can re-use variables, snippets of code, entire pages, and more. In DITA speak, this includes conref and keyref. See Content reuse (page 50) for more details.
Markdown	Yes	You can author content using Markdown syntax, specifically kramdown . This is a wiki-like syntax for HTML that you can probably pick up in 10 minutes. Where Markdown falls short, you can use HTML. Where HTML falls short, you use Liquid, which is a scripting that allows you to incorporate more advanced logic.
Responsive design	Yes	Uses Bootstrap framework for responsive design.
Translation	Yes	To translate content, send the generated HTML to your translation group. You can translate the Markdown source if your translator accepts the format, but usually Markdown is problematic. Note that this theme isn't structured well to accommodate translation projects.

Features	Supported	Notes
Collaboration	Yes	You collaborate with Jekyll projects the same way that developers collaborate with software projects. (You don't need a CMS.) Because you're working with text file formats, you can use any version control software (Git, Mercurial, Perforce, Bitbucket, etc.) as a CMS for your files.
Scalability	Yes	Your site can scale to any size. It's up to you to determine how you will design the information architecture for your pages. You can choose what you display at first, second, third, fourth, and more levels, etc. Note that when your project has thousands of pages, the build time will be longer (maybe 1 minute per thousand pages?). It really depends on how many for loops you have iterating through the pages. I recommend that you use smaller repos in your content architecture.
Lightweight architecture	Yes	You don't need a LAMP stack (Linux, Apache, MySQL, PHP) architecture to get your site running. All of the building is done on your own machine, and you then push the static HTML files onto a server.
Skinnability	Yes	You can skin your Jekyll site to look identical to pretty much any other site online. If you have a UX team, they can really skin and design the site using all the tools familiar to the modern designer – JavaScript, HTML5, CSS, jQuery, and more. Jekyll is built on the modern web development stack rather than the XML stack (XSLT, XPath, XQuery). See this tutorial for details on how to create your own Jekyll theme.
Support	Yes	The community for your Jekyll site isn't so much other tech writers (as is the case with DITA) but rather the wider web development community. Jekyll Talk is a great resource. So is Stack Overflow . See the Getting Help section of Jekyll.

Features	Supported	Notes
Blogging features	Yes	There is a simple blogging feature. This appears as news (page 0) and is intended to promote news that applies across products.
Versioning	Yes	Jekyll doesn't version your files. You upload your files to a version control system such as Github. Your files are versioned there.
PC platform	Yes	Jekyll runs on Windows. Although the experience working on the command line is better on a Mac, Windows also works, especially now that Jekyll 3.0 dropped dependencies on Python, which wasn't available by default on Windows.
jQuery plug-ins	Yes	You can use any jQuery plugins you and other JavaScript, CMS, or templating tools. However, note that if you use Ruby plugins, you can't directly host the source files on Github Pages because Github Pages doesn't allow Ruby plugins. Instead, you can just push your output to any web server. If you're not planning to use Github Pages, there are no restrictions on any plugins of any sort. Jekyll makes it super easy to integrate every kind of plugin imaginable. This theme doesn't actually use any plugins, so you can publish on Github if you want.
Bootstrap integration	Yes	This theme is built on Bootstrap . If you don't know what Bootstrap is, basically this means there are hundreds of pre-built components, styles, and other elements that you can simply drop into your site. For example, the responsive quality of the site comes about from the Bootstrap code base.

Features	Supported	Notes
Fast-loading pages	Yes	This is one of the Jekyll's strengths. Because the files are static, they loading extremely fast, approximately 0.5 seconds per page. You can't beat this for performance. (A typically database-driven site like WordPress averages about 2.5 + seconds loading time per page.) Because the pages are all static, it means they are also extremely secure. You won't get hacked like you might with a WordPress site.
Themes	Yes	You can have different themes for different outputs. If you know CSS, theming both the web and print outputs is pretty easy.
Open source	Yes	This theme is entirely open source. Every piece of code is open, viewable, and editable. Note that this openness comes at a price — it's easy to make changes that break the theme or otherwise cause errors.
Offline viewing	Yes	This theme uses relative linking throughout, so you can view the content offline and on any web-server without configuring urls and baseurls in your configuration file.

Features not available

The following features are not available.

Features	Supported	Notes
CMS interface	No	Unlike with WordPress, you don't log into an interface and navigate to your files. You work with text files and preview the site dynamically in your browser. Don't worry – this is part of the simplicity that makes Jekyll awesome. I recommend using WebStorm as your text editor.

Features	Supported	Notes
WYSIWYG interface	No	I use WebStorm to author content, because I like working in text file formats. But you can use any Markdown editor you want (e.g., Lightpaper for Mac, Marked) to author your content.
Different outputs	No	This theme provides a single website output that contains documentation for multiple products. Unlike previous iterations of the theme, it's not intended to support different outputs from the same content. However, you can easily set things up to do this by simply creating multiple configuration files and running different builds for each configuration file.
Robust search	No	The search feature is a simplistic JSON search. For more robust search, you should integrate Swiftype or Algolia. However, those services aren't currently integrated into the theme.
Standardized templates	No	You can create pages with any structure you want. The theme does not enforce topic types such as a task or concept as the DITA specification does.
Integration with Swagger	No	You can link to a SwaggerUI output, but there is no built-in integration of SwaggerUI into this documentation theme.
Templates for endpoints	No	Although static site generators work well with API documentation, there aren't any built-in templates specific to endpoints in this theme. You could construct your own, though.
eBook output	No	There isn't an eBook output for the content.

About the theme's author

Summary: I have used this theme for projects that I've worked on as a professional technical writer.

My name is Tom Johnson, and I'm a technical writer, blogger, and podcaster based in San Jose, California. For more details, see my [technical writing blog](#) and my [course on API documentation](#). See [my blog's about page](#) for more details about me.

I have used this theme and variations of it for various documentation projects. This theme has undergone several major iterations, and now it's fairly stable and full of all the features that I need. You are welcome to use it for your documentation projects for free.

I think this theme does pretty much everything that you can do with something like OxygenXML, but without the constraints of structured authoring. Everything is completely open and changeable, so if you start tinkering around with the theme's files, you can break things. But it's completely empowering as well!

With a completely open architecture and code base, you can modify the code to make it do exactly what you want, without having to jump through all kinds of confusing or proprietary code.

If there's a feature you need but it isn't available here, let me know and I might add it. Alternatively, if you fork the theme, I would love to see your modifications and enhancements. Thanks for using Jekyll.

Support

Summary: Contact me for any support issues.

Release notes 6.0

Summary: Version 6.0 of the Documentation theme for Jekyll, released July 4, 2016, implements relative links so you can view the files offline or on any server without configuring urls and baseurls. Additionally, you can store pages in subdirectories. Templates for alerts and images are available.

Relative links

You can now view the site offline rather than solely through the Jekyll preview server or deployed on a web server. The linking approach in both the sidebar and with inline links uses relative linking throughout.

Subfolders for pages

You can create folders and subfolders for your pages, similar to how you can store posts in folders and subfolders. When Jekyll builds the site, all pages get pushed into the root directory as single html files (rather than being pushed inside folders, or remaining in subfolders). See [Pages \(page 32\)](#) for more details.

Alerts templates

You can use include templates for notes, tips, and warnings. These include templates make it easier to insert notes. If you make an error, you're immediately made aware since the site won't build. See [Alerts \(page 83\)](#) for more details.

Image templates

Similar to alerts, images also have include templates. You can insert both regular images and inline images, such as images that are a button or icon. See [Images \(page 98\)](#) for more details.

Automated links using Markdown formatting

Instead of using YAML references to handle links, I've switched to a Markdown reference style approach. A links.html file iterates through the sidebar files and formats the content in the Markdown reference. You then just use Markdown syntax for the links. See [Links \(page 105\)](#) for more details.

Workflow maps

If you want to display a workflow map for a process, you can do so by adding some properties in your frontmatter. The workflow map helps guide users through a process. Both simple and complex workflow maps are available. For more details, see [Workflow maps \(page 117\)](#).

Upgrading

If you want to upgrade from an earlier version of the theme, I recommend that you download the new theme and copy of your Markdown files into the new theme. You'll then need to make adjustments to your page frontmatter, to the sidebar table of contents, links, image references, and alert references. In short, there's no easy upgrade path. But all of this won't take too long if you don't have mountains of content.

Release notes 5.0

Summary: Version 5.0 of the Documentation theme for Jekyll changes some fundamental ways the theme works to provide product-specific sidebars, intended to accommodate a site where multiple products are grouped together on the same site rather than generated out as separate outputs.

Unique sidebars for each product

In previous versions of the theme, I built the theme to generate different outputs for different scenarios based on various filtering attributes that might include product, version, platform, and audience variants.

However, this model results in siloed outputs and lots of separate file directories to manage. Instead of having 30 separate sites for your content (or however many variants you might have been producing), in this version of the theme I've moved towards a strategy of having one site with multiple products.

For each product, you can associate a unique sidebar with each of the product's pages. This allows you to have all your documentation on one site, but with separate navigation that is tailored to a view of that product.

You can still output to both web and PDF. And if you really need multiple site outputs, you can still do so by using multiple configuration files that trigger different builds. But my conclusion after using the multiple site output model for some years is that it's a bad practice for tech comm.

Permalinks

With this theme, since you'll be publishing to one site, I've implement permalinks instead of relative links. Using permalinks means the way you store pages is much more flexible. You can store topics in folders and subfolders, etc., to any degree. But note that with permalinks you can't view the content offline (outside of Jekyll's preview server) nor on a separate site other than the one specified in the configuration file. Permalinks are how Jekyll was designed to work, and the sites just work better that way.

Kramdown and Rouge

I also switched from redcarpet and Pygments to Kramdown and Rouge to align with the current direction of Jekyll 3.0. Kramdown is a Markdown filter (it's slightly different from Github-flavored Markdown). Rouge is a syntax highlighter. Pygments had some dependencies on Python, which made it more cumbersome for Windows users.

Blog feature

I included a blog feature with this version of the theme. You can write posts and view them through the News link. There's also an archive for blog posts that sorts posts by year.

Additionally, the tagging system works across both the blog and pages, so your tags allow users to move laterally across the site based on topics they're interested in. When you view a tag archive, the sidebar shows a list of tags.

Updated documentation

I updated the documentation for the theme. The switch from the multi-site outputs to the single-site with multiple sidebars required updating a lot of different parts of the documentation and code.

Fixed errors

Previously I had some errors with the HTML that showed up in w3c HTML validator analyses. This caused some small problems in certain browsers or systems less tolerant of the errors. I fixed all of the errors.

Accessing the old theme

If you want to access the old theme, you can still find it [here](#).

About Ruby, Gems, Bundler, and other prerequisites

Summary: Ruby is a programming language you must have on your computer in order to build Jekyll locally. Ruby has various gems (or plugins) that provide various functionality. Each Jekyll project usually requires certain gems.

About Ruby

Jekyll runs on Ruby, a programming language. You have to have Ruby on your computer in order to run Ruby-based programs like Jekyll. Ruby is installed on the Mac by default, but you must add it to Windows.

About Ruby Gems

Ruby has a number of plugins referred to as “gems.” Just because you have Ruby doesn’t mean you have all the necessary Ruby gems that your program needs to run. Gems provide additional functionality for Ruby programs. There are thousands of [Rubygems](#) available for you to use.

Some gems depend on other gems for functionality. For example, the Jekyll gem might depend on 20 other gems that must also be installed.

Each gem has a version associated with it, and not all gem versions are compatible with each other.

Rubygem package managers

[Bundler](#) is a gem package manager for Ruby, which means it goes out and gets all the gems you need for your Ruby programs. If you tell Bundler you need the [jekyll gem](#), it will retrieve all the dependencies on the jekyll gem as well – automatically.

Not only does Bundler retrieve the right gem dependencies, but it’s smart enough to retrieve the right versions of each gem. For example, if you get the [github-pages](#) gem, it will retrieve all of these other gems:

```
github-pages-health-check = 1.1.0
jekyll = 3.0.3
jekyll-coffeescript = 1.0.1
jekyll-feed = 0.4.0
jekyll-gist = 1.4.0
jekyll-github-metadata = 1.9.0
jekyll-mentions = 1.1.2
jekyll-paginate = 1.1.0
jekyll-redirect-from = 0.10.0
jekyll-sass-converter = 1.3.0
jekyll-seo-tag = 1.3.2
jekyll-sitemap = 0.10.0
jekyll-textile-converter = 0.1.0
jemoji = 0.6.2
kramdown = 1.10.0
liquid = 3.0.6
mercenary ~> 0.3
rdiscount = 2.1.8
redcarpet = 3.3.3
RedCloth = 4.2.9
rouge = 1.10.1
terminal-table ~> 1.
```

See how Bundler retrieved version 3.0.3 of the jekyll gem, even though (as of this writing) the latest version of the jekyll gem is 3.1.2? That's because github-pages is only compatible up to jekyll 3.0.3. Bundler handles all of this dependency and version compatibility for you.

Trying to keep track of which gems and versions are appropriate for your project can be a nightmare. This is the problem Bundler solves. As explained on [Bundler.io](https://bundler.io):

Bundler provides a consistent environment for Ruby projects by tracking and installing the exact gems and versions that are needed.

Bundler is an exit from dependency hell, and ensures that the gems you need are present in development, staging, and production. Starting work on a project is as simple as `bundle install`.

Gemfiles

Bundler looks in a project's "Gemfile" (no file extension) to see which gems are required by the project. The Gemfile lists the source and then any gems, like this:

```
source "https://rubygems.org"

gem 'github-pages'
gem 'jekyll'
```

The source indicates the site where Bundler will retrieve the gems:

<https://rubygems.org>.

The gems it retrieves are listed separately on each line.


Here no versions are specified. Sometimes gemfiles will specify the versions like this:

```
gem 'kramdown', '1.0'
```

This means Bundler should get version 1.0 of the kramdown gem.

To specify a subset of versions, the Gemfile looks like this:

```
gem 'jekyll', '~> 2.3'
```

The  sign means greater than or equal to the *last digit before the last period in the number*.

Here it will get any gem equal to 2.3 but less than 3.0.

If it adds another digit, the scope is affected:

```
gem 'jekyll', '~>2.3.1'
```

This means to get any gem equal to 2.3.1 but less than 2.4.

If it looks like this:

```
gem 'jekyll', '~> 3.0', '>= 3.0.3'
```

This will get any Jekyll gem between versions 3.0 and up to 3.0.3.

See this [Stack Overflow post](#) for more details.

Gemfile.lock

After Bundler retrieves and installs the gems, it makes a detailed list of all the gems and versions it has installed for your project. The snapshot of all gems + versions installed is stored in your Gemfile.lock file, which might look like this:

GEM

```
remote: https://rubygems.org/
specs:
  RedCloth (4.2.9)
  activesupport (4.2.5.1)
    i18n (~> 0.7)
    json (~> 1.7, >= 1.7.7)
    minitest (~> 5.1)
    thread_safe (~> 0.3, >= 0.3.4)
    tzinfo (~> 1.1)
  addressable (2.3.8)
  coffee-script (2.4.1)
    coffee-script-source
    execjs
  coffee-script-source (1.10.0)
  colorator (0.1)
  ethon (0.8.1)
    ffi (>= 1.3.0)
  execjs (2.6.0)
  faraday (0.9.2)
    multipart-post (>= 1.2, < 3)
  ffi (1.9.10)
  gemoji (2.1.0)
  github-pages (52)
    RedCloth (= 4.2.9)
    github-pages-health-check (= 1.0.1)
    jekyll (= 3.0.3)
    jekyll-coffeescript (= 1.0.1)
    jekyll-feed (= 0.4.0)
    jekyll-gist (= 1.4.0)
    jekyll-mentions (= 1.0.1)
    jekyll-paginate (= 1.1.0)
    jekyll-redirect-from (= 0.9.1)
    jekyll-sass-converter (= 1.3.0)
    jekyll-seo-tag (= 1.3.1)
    jekyll-sitemap (= 0.10.0)
    jekyll-textile-converter (= 0.1.0)
    jemoji (= 0.5.1)
    kramdown (= 1.9.0)
    liquid (= 3.0.6)
    mercenary (~> 0.3)
    rdiscount (= 2.1.8)
    redcarpet (= 3.3.3)
    rouge (= 1.10.1)
    terminal-table (~> 1.4)
```

```
github-pages-health-check (1.0.1)
  addressable (~> 2.3)
  net-dns (~> 0.8)
  octokit (~> 4.0)
  public_suffix (~> 1.4)
  typhoeus (~> 0.7)
html-pipeline (2.3.0)
  activesupport (>= 2, < 5)
  nokogiri (>= 1.4)
i18n (0.7.0)
jekyll (3.0.3)
  colorator (~> 0.1)
  jekyll-sass-converter (~> 1.0)
  jekyll-watch (~> 1.1)
  kramdown (~> 1.3)
  liquid (~> 3.0)
  mercenary (~> 0.3.3)
  rouge (~> 1.7)
  safe_yaml (~> 1.0)
jekyll-coffeescript (1.0.1)
  coffee-script (~> 2.2)
jekyll-feed (0.4.0)
jekyll-gist (1.4.0)
  octokit (~> 4.2)
jekyll-mentions (1.0.1)
  html-pipeline (~> 2.3)
  jekyll (~> 3.0)
jekyll-paginate (1.1.0)
jekyll-redirect-from (0.9.1)
  jekyll (>= 2.0)
jekyll-sass-converter (1.3.0)
  sass (~> 3.2)
jekyll-seo-tag (1.3.1)
  jekyll (~> 3.0)
jekyll-sitemap (0.10.0)
jekyll-textile-converter (0.1.0)
  RedCloth (~> 4.0)
jekyll-watch (1.3.1)
  listen (~> 3.0)
jemoji (0.5.1)
  gemoji (~> 2.0)
  html-pipeline (~> 2.2)
  jekyll (>= 2.0)
json (1.8.3)
kramdown (1.9.0)
```

```
liquid (3.0.6)
listen (3.0.6)
  rb-fsevent (>= 0.9.3)
  rb-inotify (>= 0.9.7)
mercenary (0.3.5)
mini_portile2 (2.0.0)
minitest (5.8.4)
multipart-post (2.0.0)
net-dns (0.8.0)
nokogiri (1.6.7.2)
  mini_portile2 (~> 2.0.0.rc2)
octokit (4.2.0)
  sawyer (~> 0.6.0, >= 0.5.3)
public_suffix (1.5.3)
rb-fsevent (0.9.7)
rb-inotify (0.9.7)
  ffi (>= 0.5.0)
rdiscount (2.1.8)
redcarpet (3.3.3)
rouge (1.10.1)
safe_yaml (1.0.4)
sass (3.4.21)
sawyer (0.6.0)
  addressable (~> 2.3.5)
  faraday (~> 0.8, < 0.10)
terminal-table (1.5.2)
thread_safe (0.3.5)
typhoeus (0.8.0)
  ethon (>= 0.8.0)
tzinfo (1.2.2)
  thread_safe (~> 0.1)
```

PLATFORMS

ruby

DEPENDENCIES

github-pages
jekyll

BUNDLED WITH

1.11.2

You can always delete the Gemlock file and run Bundle install again to get the latest versions. You can also run `bundle update`, which will ignore the Gemlock file to get the latest versions of each gem.

To learn more about Bundler, see [Bundler's Purpose and Rationale](#) .

Install Jekyll on Mac

Summary: Installation of Jekyll on Mac is usually less problematic than on Windows. However, you may run into permissions issues with Ruby that you must overcome. You should also use Bundler to be sure that you have all the required gems and other utilities on your computer to make the project run.

Ruby and RubyGems

Ruby and [RubyGems](#) are usually installed by default on Macs. Open your Terminal and type `which ruby` and `which gem` to confirm that you have Ruby and Rubygems. You should get a response indicating the location of Ruby and Rubygems.

If you get responses that look like this:

```
/usr/local/bin/ruby
```

and

```
/usr/local/bin/gem
```

Great! Skip down to the [Bundler \(page 26\)](#) section.

However, if your location is something like `/Users/MacBookPro/.rvm/rubies/ruby-2.2.1/bin/gem`, which points to your system location of Rubygems, you will likely run into permissions errors when trying to get a gem. A sample permissions error (triggered when you try to install the jekyll gem such as `gem install jekyll`) might look like this for Rubygems:

```
>ERROR: While executing gem ... (Gem::FilePermissionError)
  You don't have write permissions for the /Library/Ruby/Gems/
  2.0.0 directory.
```

Instead of changing the write permissions on your operating system's version of Ruby and Rubygems (which could pose security issues), you can install another instance of Ruby (one that is writable) to get around this.

Install Homebrew

Homebrew is a package manager for the Mac, and you can use it to install an alternative instance of Ruby code. To install Homebrew, run this command:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

If you already had Homebrew installed on your computer, be sure to update it:

```
brew update
```

Install Ruby through Homebrew

Now use Homebrew to install Ruby:

```
brew install ruby
```

Log out of terminal, and then then log back in.

When you type `which ruby` and `which gem`, you should get responses like this:

```
/usr/local/bin/ruby
```

And this:

```
/usr/local/bin/gem
```

Now Ruby and Rubygems are installed under your username, so these directories are writeable.

Note that if you don't see these paths, try restarting your computer or try installing rbenv, which is a Ruby version management tool. If you still have issues getting a writeable version of Ruby, you need to resolve them before installing Bundler.

Install the Jekyll gem

At this point you should have a writeable version of Ruby and Rubygem on your machine.

Now use `gem` to install Jekyll:

```
gem install jekyll
```

You can now use Jekyll to create new Jekyll sites following the quick-start instructions on Jekyllrb.com.

Installing dependencies through Bundler

Some Jekyll themes will require certain Ruby gem dependencies. These dependencies are stored in something called a Gemfile, which is packaged with the Jekyll theme. You can install these dependencies through Bundler. (Although you don't need to install Bundler for this Documentation theme, it's a good idea to do so.)

[Bundler](#) is a package manager for RubyGems. You can use it to get all the gems (or Ruby plugins) that you need for your Jekyll project.

You install Bundler by using the gem command with RubyGems:

```
gem install bundler
```

If you're prompted to switch to superuser mode (`sudo`) to get the correct permissions to install Bundler in that directory, avoid doing this. All other applications that need to use Bundler will likely not have the needed permissions to run.

Bundler goes out and retrieves all the gems that are specified in a Jekyll project's Gemfile. If you have a gem that depends on other gems to work, Bundler will go out and retrieve all of the dependencies as well. (To learn more about Bundler, see [About Ruby Gems \(page 16\)](#)).

The vanilla Jekyll site you create through `jekyll new my-awesome-site` doesn't have a Gemfile, but many other themes (including the Documentation theme for Jekyll) do have a Gemfile.

Serve the Jekyll Documentation theme

1. Browse to the directory where you downloaded the Documentation theme for Jekyll.
2. Type `jekyll serve`
3. Go to the preview address in the browser. (Make sure you include the `/` at the end.)

Resolve “No Github API authentication” errors

After making an edit, Jekyll auto-rebuilds the site. If you have the Gemfile in the theme with the github-pages gem, you may see the following error:

```
GitHub Metadata: No GitHub API authentication could be found. Some fields may be missing or have incorrect data.
```

If you see this error, you will need to take some additional steps to resolve it. (Note that this error only appears if you have the github-pages gem in your gemfile.) The resolution involves adding a Github token and a cert file.

Note: These instructions apply to Mac OS X, but they're highly similar to Windows. These instructions are adapted from a post on [Knight Codes](#). If you're on Windows, see the Knight Codes post for details instead of following along below.

To resolve the “No Github API authentication” error:

1. Follow Github's instructions to [create a personal access token](#).
2. Open the `.bash_profile` file in your user directory:

```
open ~/.bash_profile
```

The file will open in your default terminal editor. If you don't have a `.bash_profile` file, you can just create a file with this name. Note that files that begin with `.` are hidden, so if you're looking in your user directory for the file, use `ls -a` to see hidden files.

3. In your `.bash_profile` file, reference your token as a system variable like this:

```
export JEKYLL_GITHUB_TOKEN=abc123abc123abc123abc123abc123abc123abc123abc123
```

Replace `abc123...` with your own token that you generated in step 1.

4. Go to [\[https://curl.haxx.se/ca/cacert.pem\]](https://curl.haxx.se/ca/cacert.pem)[\[https://curl.haxx.se/ca/cacert.pem\]](https://curl.haxx.se/ca/cacert.pem). **Right-click the page, select `**Save as`**, and save the file on your computer (save it somewhere safe, where you won't delete it). Name the file `cacert`.
5. Open your `.bash_profile` file again and add this line, replacing `Users/johndoe/projects/` with the path to your `cacert.pem` file:

```
export SSL_CERT_FILE=/Users/johndoe/projects/cacert.pem
```

6. Close and restart your terminal.

Browse to your jekyll project and run `bundle exec jekyll serve`. Make an edit to a file and observe that no Github API errors appear when Jekyll rebuilds the project.

Install Jekyll on Windows

✓ **Tip:** For a better terminal emulator on Windows, use [Git Bash](#) . Git Bash gives you Linux-like control on Windows.

Install Ruby and Ruby Development Kit

First you must install Ruby because Jekyll is a Ruby-based program and needs Ruby to run.

1. Go to [RubyInstaller for Windows](#) .
2. Under **RubyInstallers**, download and install one of the Ruby installers under the **WITH DEVKIT** list (usually the recommended/highlighted option).
3. Double-click the downloaded file and proceed through the wizard to install it. Run the `ridk install` step on the last stage of the installation wizard.
4. Open a new command prompt window or Git Bash session.

Install the Jekyll gem

At this point you should have Ruby and Rubygem on your machine.

Now use `gem` to install Jekyll:

```
gem install jekyll
```

You can now use Jekyll to create new Jekyll sites following the quick-start instructions on [Jekyllrb.com](#) .

Installing dependencies through Bundler

Some Jekyll themes will require certain Ruby gem dependencies. These dependencies are stored in something called a Gemfile, which is packaged with the Jekyll theme. You can install these dependencies through Bundler. (Although you don't need to install Bundler for this Documentation theme, it's a good idea to do so.)

[Bundler](#) is a package manager for RubyGems. You can use it to get all the gems (or Ruby plugins) that you need for your Jekyll project.

You install Bundler by using the gem command with RubyGems:

Install Bundler

1. Browse to the directory where you downloaded the Documentation theme for Jekyll.
2. Delete or rename the existing `Gemfile` and `Gemfile.lock` files.
3. Install Bundler: `gem install bundler`
4. Initialize Bundler: `bundle init`

This will create a new Gemfile.

5. Open the Gemfile in a text editor.

Typically you can open files from the Command Prompt by just typing the filename, but because Gemfile doesn't have a file extension, no program will automatically open it. You may need to use your File Explorer and browse to the directory, and then open the Gemfile in a text editor such as Notepad.

6. Remove the existing contents. Then paste in the following:

```
source "https://rubygems.org"

gem 'wdm'
gem 'jekyll'
```

The [wdm gem](#) allows for the polling of the directory and rebuilding of the Jekyll site when you make changes. This gem is needed for Windows users, not Mac users.

7. Save and close the file.
8. Type `bundle install`.

Bundle retrieves all the needed gems and gem dependencies and downloads them to your computer. At this time, Bundle also takes a snapshot of all the gems used in your project and creates a Gemfile.lock file to store this information.

Git Clients for Windows

Although you can use the default command prompt with Windows, it's recommended that you use [Git Bash](#) instead. The Git Bash client will allow you to run shell scripts and execute other Unix commands.

Serve the Jekyll Documentation theme

1. Browse to the directory where you downloaded the Documentation theme for Jekyll.
2. Type `jekyll serve`
3. Go to the preview address in the browser. (Make sure you include the `/` at the end.)

Unfortunately, the Command Prompt doesn't allow you to easily copy and paste the URL, so you'll have to type it manually.

Resolving Github Metadata errors

After making an edit, Jekyll auto-rebuilds the site. If you have the Gemfile in the theme with the github-pages gem, you may see the following error:

```
GitHub Metadata: No GitHub API authentication could be found. Some fields may be missing or have incorrect data.
```

If so, you will need to take some additional steps to resolve it. (Note that this error only appears if you have the github-pages gem in your gemfile.) The resolution involves adding a Github token and a cert file.

See this post on [Knight Codes](#) for instructions on how to fix the error. You basically generate a personal token on Github and set it as a system variable. You also download a certification file and set it as a system variable. This resolves the issue.

Pages

Summary: This theme primarily uses pages. You need to make sure your pages have the appropriate frontmatter. One frontmatter tag your users might find helpful is the summary tag. This functions similar in purpose to the shortdesc element in DITA.

Where to author content

Use a text editor such as Sublime Text, WebStorm, IntelliJ, or Atom to create pages. Atom is recommended because it's created by Github, which is driving some of the Jekyll development through Github Pages.

Where to save pages

You can store your pages in any folder structures you want, with any level of folder nesting. The site output will pull all of those pages out of their folders and put them into the root directory. Check out the `_site` folder, which is where Jekyll is generated, to see the difference between your project's structure and the resulting site output.

The listing of all pages in the root directory (which happens when you add a permalink property to the pages) is what allows the relative linking and offline viewing of the site to work.

Frontmatter

Make sure each page has frontmatter at the top like this:

```
---
title: Alerts
tags: [formatting]
keywords: notes, tips, cautions, warnings, admonitions
last_updated: July 3, 2016
summary: "You can insert notes, tips, warnings, and important alerts in your content."
sidebar: mydoc_sidebar
permalink: mydoc_alerts.html
---
```

Frontmatter is always formatted with three hyphens at the top and bottom. Your frontmatter must have a `title` and `permalink` value. All the other values are optional.

Note that you cannot use variables in frontmatter.

The following table describes each of the frontmatter that you can use with this theme:

Frontmatter	Required?	Description
title	Required	The title for the page
tags	Optional	Tags for the page. Make all tags single words, with underscores if needed (rather than spaces). Separate them with commas. Enclose the whole list within brackets. Also, note that tags must be added to <code>_data/tags_doc.yml</code> to be allowed entrance into the page. This prevents tags from becoming somewhat random and unstructured. You must create a tag page for each one of your tags following the pattern shown in the tags folder. (Tag pages aren't automatically created.)
keywords	Optional	Synonyms and other keywords for the page. This information gets stuffed into the page's metadata to increase SEO. The user won't see the keywords, but if you search for one of the keywords, it will be picked up by the search engine.
last_updated	Optional	The date the page was last updated. This information could be helpful for readers trying to evaluate how current and authoritative information is. If included, the <code>last_updated</code> date appears in the footer of the page in small font.
sidebar	Required	Refers to the sidebar data file for this page. Don't include the <code>".yml"</code> file extension for the sidebar — just provide the file name. If no sidebar is specified, this value will inherit the <code>default</code> property set in your <code>_config.yml</code> file for the page's frontmatter.

Frontmatter	Required?	Description
summary	Optional	A 1-2 word sentence summarizing the content on the page. This gets formatted into the summary section in the page layout. Adding summaries is a key way to make your content more scannable by users (check out Jakob Nielsen's site for a great example of page summaries.) The only drawback with summaries is that you can't use variables in them.
permalink	Required	The permalink <i>must</i> match the filename in order for automated links to work. Additionally, you must include the ".html" in the filename. Do not put forward slashes around the permalink (this makes Jekyll put the file inside a folder in the output). When Jekyll builds the site, it will put the page into the root directory rather than leaving it in a subdirectory or putting it inside a folder and naming the file index.html. Having all files flattened in the root directory is essential for relative linking to work and for all paths to JS and CSS files to be valid.
datatable	Optional	'true'. If you add <code>datatable: true</code> in the frontmatter, scripts for the jQuery Datatables plugin get included on the page. You can see the scripts that conditionally appear by looking in the <code>_layouts/default.html</code> page.
toc	Optional	If you specify <code>toc: false</code> in the frontmatter, the page won't have the table of contents that appears below the title. The toc refers to the list of jump links below the page title, not the sidebar navigation. You probably want to hide the TOC on the homepage and product landing pages.

Colons in page titles

If you want to use a colon in your page title, you must enclose the title's value in quotation marks.

Page names and excluding files from outputs

By default, everything in your project is included in the output. You can exclude all files that don't belong to that project by specifying the file name, the folder name, or by using wildcards in your configuration file:

```
exclude:  
  
- filename.md  
- subfolder_name/  
- mydoc_*  
- gitignore
```

Wildcards will exclude every match after the `*`.

Saving pages as drafts

If you add `published: false` in the frontmatter, your page won't be published. You can also move draft pages into the `_drafts` folder to exclude them from the build. With posts, you can also keep them as drafts by omitting the date in the title.

Markdown or HTML format

Pages can be either Markdown or HTML format (specified through either an `.md` or `.html` file extension).

If you use Markdown, you can also include HTML formatting where needed. But if your format is HTML, you must add a `markdown="1"` attribute to the element in order to use Markdown inside that HTML element:

```
<div markdown="1">This is a [link](http://exmaple.com).</div>
```

For your Markdown files, note that a space or two indent will set text off as code or blocks, so avoid spacing indents unless intentional.

If you have a lot of HTML, as long as the top and bottom tags of the HTML are flush left in a Markdown file, all the tags inside those bookend HTML tags will render as HTML, regardless of their indentation. (This can be especially useful for tables.)

Page names

I recommend prefixing your page names with the product, such as “mydoc_pages” instead of just “pages.” This way if you have other products that also have topics with generic names such as “pages,” there won’t be naming conflicts.

Additionally, consider adding the product name in parentheses after the title, such as “Pages (Mydoc)” so that users can clearly navigate different topics for each product.

Kramdown Markdown

Kramdown is the Markdown flavor used in the theme. This mostly aligns with Github-flavored Markdown, but with some differences in the indentation allowed within lists. Basically, Kramdown requires you to line up the indent between list items with the first starting character after the space in your list item numbering. See this [blog post on Kramdown and Rouge](#) for more details.

You can use standard Multimarkdown syntax for tables. You can also use fenced code blocks with lexers specifying the type of code. The configuration file shows the Markdown processor and extension:

```
highlighter: rouge
markdown: kramdown
kramdown:
  input: GFM
  auto_ids: true
  hard_wrap: false
  syntax_highlighter: rouge
```

Automatic mini-TOCs

By default, a TOC appears at the top of your pages and posts. If you don’t want the TOC to appear for a specific page, such as for a landing page or other homepage, add `toc: false` in the frontmatter of the page.

The mini-TOC requires you to use the `##` Markdown syntax for headings. If you use `<h2>` elements, you must add an ID attribute for the heading element in order for it to appear in the mini-TOC (for example, `<h2 id="mysampleid">Heading</h2>`).

Headings

Use pound signs before the heading title to designate the level. Note that kramdown requires headings to have one space before and after the heading. Without this space above and below, the heading won't render into HTML.

```
## Second-level heading
```

Result:

Second-level heading

```
### Third-level heading
```

Result:

Third-level heading

```
#### Fourth-level heading
```

Result:

Fourth-level heading

Headings with ID Tags

If you want to use a specific ID tag with your heading, add it like this:

```
## Headings with ID Tags {#someIdTag}
```

Then you can reference it with a link like this on the same page:

```
[Some link](#someIdTag)
```

Result:

[Some link \(page 37\)](#)

For details about linking to headings on different pages, see [Automated links to headings on pages \(page 0\)](#).

Specify a particular page layout

The configuration file sets the default layout for pages as the “page” layout.

You can create other layouts inside the layouts folder. If you create a new layout, you can specify that your page use your new layout by adding `layout: mylayout.html` in the page’s frontmatter. Whatever layout you specify in the frontmatter of a page will override the layout default set in the configuration file.

Comments

Disqus, a commenting system, is integrated into the theme. In the configuration file, specify the Disqus code for the universal code, and Disqus will appear. If you don’t add a Disqus value, the Disqus form isn’t included.

Posts

Summary: You can use posts when you want to create blogs or news type of content.

About posts

Posts are typically used for blogs or other news information because they contain a date and are sorted in reverse chronological order.

You create a post by adding a file in the `_posts` folder that is named `yyyy-mm-dddd-permalink.md`, which might be `2016-02-25-my-latest-updates.md`. You can use any number of subfolders here that you want.

Posts use the `post.html` layout in the `_layouts` folder when you are viewing the post.

The `news.html` file in the root directory shows a reverse chronological listing of the 10 latest posts

Allowed frontmatter

The frontmatter you can use with posts is as follows:

```
---
title: My sample post
tags: content_types
keywords: pages, authoring, exclusion, frontmatter
sidebar: mydoc_sidebar
permalink: mydoc_pages.html
summary: "This is some summary frontmatter for my sample post."
---
```

Frontmatter	Required?	Description
title	Required	The title for the page

Frontmatter	Required?	Description
tags	Optional	Tags for the page. Make all tags single words, with underscores if needed. Separate them with commas. Enclose the whole list within brackets. Also, note that tags must be added to <code>_data/tags_doc.yml</code> to be allowed entrance into the page. This prevents tags from becoming somewhat random and unstructured. You must create a tag page for each one of your tags following the sample pattern in the <code>tags</code> folder. (Tag pages aren't automatically created.)
keywords	Optional	Synonyms and other keywords for the page. This information gets stuffed into the page's metadata to increase SEO. The user won't see the keywords, but if you search for one of the keywords, it will be picked up by the search engine.
sidebar	Required	Refers to the sidebar data file for this page. Don't include the <code>.yml</code> file extension for the sidebar — just provide the file name. If no sidebar is specified, this value will inherit the <code>default</code> property set in your <code>_config.yml</code> file for the page's frontmatter.
permalink	Required	This theme uses permalinks to facilitate the linking. You specify the permalink want for the page, and the <code>_site</code> output will put the page into the root directory when you publish. Follow the same convention here as you do with page permalinks — list the file name followed by the <code>.html</code> extension.
summary	Optional	A 1-2 word sentence summarizing the content on the page. This gets formatted into the summary section in the page layout. Adding summaries is a key way to make your content more scannable by users (check out Jakob Nielsen's site for a great example of page summaries.) The only drawback with summaries is that you can't use variables in them.

Lists

Summary: This page shows how to create both bulleted and numbered lists

Bulleted Lists

This is a bulleted list:

```
* first item  
* second item  
* third item
```

Result:

- first item
- second item
- third item

Numbered list

This is a simple numbered list:

```
1. First item.  
1. Second item.  
1. Third item.
```

Result:

1. First item.
2. Second item.
3. Third item.

You can use whatever numbers you want — when the Markdown filter processes the content, it will assign the correct numbers to the list items.

Complex Lists

Here's a more complex list:

1. Sample first item.
 - * sub-bullet one
 - * sub-bullet two
2. Continuing the list
 1. sub-list numbered one
 2. sub-list numbered two
3. Another list item.

Result:

1. Sample first item.
 - sub-bullet one
 - sub-bullet two
2. Continuing the list
 - a. sub-list numbered one
 - b. sub-list numbered two
3. Another list item.

Another Complex List

Here's a list with some intercepting text:

1. Sample first item.

This is a result statement that talks about something....

2. Continuing the list

```
<div markdown="span" class="alert alert-info" role="aler
t"><i class="fa fa-info-circle"></i> <b>Note:</b> Remember to d
o this. If you have "quotes", you must escape them.</div>
```

Here's a list in here:

- * first item
- * second item

3. Another list item.

```
```js
function alert("hello");
```
```

4. Another item.

Result:

1. Sample first item.

This is a result statement that talks about something....

2. Continuing the list

Note: Remember to do this. If you have “quotes”, you must escape them.

Here's a list in here:

- first item
- second item

3. Another list item.

```
function alert("hello");
```

4. Another item.

Key Principle to Remember with Lists

The key principle is to line up the first character after the dot following the number:

```

61
62 Here's a list with some intercepting text:
63
64 1. Sample first item.
65
66   This is a result statement that talks about something...
67
68 2. Continuing the list
69
70   {% include note.html content="Remember to do this. If you have \"quotes\", you must escape
71
72   Here's a list in here:
73
74   * first item
75   * second item
76
77 3. Another list item.
78
79   ```js
80   function alert("hello");
81   ```
82
83 4. Another item.
84
85 The key principle is to line up |
86
87 ## Links
88

```

Lining up the left edge ensures the list stays in tact.

For the sake of simplicity, use two spaces after the dot for numbers 1 through 9. Use one space for numbers 10 and up. If any part of your list doesn't align symmetrically on this left edge, the list will not render correctly. Also note that this is characteristic of kramdown-flavored Markdown and may not yield the same results in other Markdown flavors.

Conditional logic

Summary: You can implement advanced conditional logic that includes if statements, or statements, unless, and more. This conditional logic facilitates single sourcing scenarios in which you're outputting the same content for different audiences.

About Liquid and conditional statements

If you want to create different outputs for different audiences, you can do all of this using a combination of Jekyll's Liquid markup and values in your configuration file. This is how I previously configured the theme. I had different configuration files for each output. Each configuration file specified different values for product, audience, version, and so on. Then I had different build processes that would leverage the different configuration files. It seemed like a perfect implementation of DITA-like techniques with Jekyll.

But I soon found that having lots of separate outputs for a project was undesirable. If you have 10 different outputs that have different nuances for different audiences, it's hard to manage and maintain. In this latest version of the theme, I consolidated all information into the same output to explicitly do away with the multi-output approach.

As such, the conditional logic won't have as much play as it previously did. Instead of conditions, you'll probably want to incorporate [navtabs \(page 0\)](#) to split up the information.

However, you can still of course use conditional logic as needed.

✓ **Tip:** Definitely check out [Liquid's documentation](#) for more details about how to use operators and other liquid markup. The notes here are a small, somewhat superficial sample from the site.

Where to store filtering values

You can filter content based on values that you have set either in your page's frontmatter, a config file, or in a file in your `_data` folder. If you set the attribute in your config file, you need to restart the Jekyll server to see the changes. If you set the value in a file in your `_data` folder or page frontmatter, you don't need to restart the server when you make changes.

Conditional logic based on config file value

Here's an example of conditional logic based on a value in the page's frontmatter. Suppose you have the following in your frontmatter:

```
platform: mac
```

On a page in my site (it can be HTML or markdown), you can conditionalize content using the following:

```
{% if page.platform == "mac" %}  
Here's some info about the Mac.  
{% elsif page.platform == "windows" %}  
Here's some info about Windows ...  
{% endif %}
```

This uses simple `if-elsif` logic to determine what is shown (note the spelling of `elsif`). The `else` statement handles all other conditions not handled by the `if` statements.

Here's an example of `if-else` logic inside a list:

```
To bake a casserole:  
  
1. Gather the ingredients.  
{% if page.audience == "writer" %}  
2. Add in a pound of meat.  
{% elsif page.audience == "designer" %}  
3. Add in an extra can of beans.  
{% endif %}  
3. Bake in oven for 45 min.
```

You don't need the `elsif` or `else`. You could just use an `if` (but be sure to close it with `endif`).

Or operator

You can use more advanced Liquid markup for conditional logic, such as an `or` command. See [Shopify's Liquid documentation](#) for more details.

For example, here's an example using `or`:

```
{% if page.audience contains "vegan" or page.audience == "vegetarian" %}  
  Then run this...  
{% endif %}
```

Note that you have to specify the full condition each time. You can't shorten the above logic to the following:

```
{% if page.audience contains "vegan" or "vegetarian" %}  
  // run this.  
{% endif %}
```

This won't work.

Unless operator

You can also use `unless` in your logic, like this:

```
{% unless site.output == "pdf" %}  
  ...do this  
{% endunless %}
```

When figuring out this logic, read it like this: “Run the code here *unless* this condition is satisfied.”.

Don't read it the other way around or you'll get confused. (It's *not* executing the code only if the condition is satisfied.)

Storing conditions in the `_data` folder

Here's an example of using conditional logic based on a value in a data file:

```
{% if site.data.options.output == "alpha" %}  
show this content...  
{% elsif site.data.options.output == "beta" %}  
show this content...  
{% else %}  
this shows if neither of the above two if conditions are met.  
{% endif %}
```

To use this, I would need to have a `_data` folder called `options` where the `output` property is stored.

Specifying the location for `_data`

You can also specify a `data_source` for your data location in your configuration file. Then you aren't limited to simply using `_data` to store your data files.

For example, suppose you have 2 projects: `alpha` and `beta`. You might store all the data files for `alpha` inside `data_alpha`, and all the data files for `beta` inside `data_beta`.

In your `alpha` configuration file, specify the data source like this:

```
data_source: data_alpha
```

Then create a folder called `_data_alpha`.

For your `beta` configuration file, specify the data source like this:

```
data_source: data_beta
```

Then create a folder called `_data_beta`.

Conditions versus includes

If you have a lot of conditions in your text, it can get confusing. As a best practice, whenever you insert an `if` condition, add the `endif` at the same time. This will reduce the chances of forgetting to close the if statement. Jekyll won't build if there are problems with the liquid logic.

If your text is getting busy with a lot of conditional statements, consider putting a lot of content into includes so that you can more easily see where the conditions begin and end.

Content reuse

Summary: You can reuse chunks of content by storing these files in the includes folder. You then choose to include the file where you need it. This works similar to conref in DITA, except that you can include the file in any content type.

About content reuse

You can embed content from one file inside another using includes. Put the file containing content you want to reuse (e.g., mypage.html) inside the `_includes/` custom folder and then use a tag like this:

```
{% include custom/mypage.html %}
```

With content in your `_includes` folder, you don't add any frontmatter to these pages because they will be included on other pages already containing frontmatter.

Also, when you include a file, all of the file's contents get included. You can't specify that you only want a specific part of the file included. However, you can use parameters with includes.

Page-level variables

You can also create custom variables in your frontmatter like this:

```
---  
title: Page-level variables  
permalink: page_level_variables/  
thing1: Joe  
thing2: Dave  
---
```

You can then access the values in those custom variables using the `page` namespace, like this:

```
thing1: {{page.thing1}}  
thing2: {{page.thing2}}
```

I use includes all the time. Most of the includes in the `_includes` directory are pulled into the theme layouts. For those includes that change, I put them inside `custom` and then inside a specific project folder.

Collections

Summary: Collections are useful if you want to loop through a special folder of pages that you make available in a content API. You could also use collections if you have a set of articles that you want to treat differently from the other content, with a different layout or format.

What are collections

Collections are custom content types different from pages and posts. You might create a collection if you want to treat a specific set of articles in a unique way, such as with a custom layout or listing. For more detail on collections, see [Ben Balter's explanation of collections here](#).

Create a collection

To create a collection, add the following in your configuration file:

```
collections:
  tooltips:
    output: true
```

In this example, “tooltips” is the name of the collection.

Interacting with collections

You can interact with collections by using the `site.collectionname` namespace, where `collectionname` is what you've configured. In this case, if I wanted to loop through all tooltips, I would use `site.tooltips` instead of `site.pages` or `site.posts`.

See [Collections in the Jekyll documentation](#) for more information.

How to use collections

I haven't found a huge use for collections in normal documentation. However, I did find a use for collections in generating a tooltip file that would be used for delivering tooltips to a user interface from text files in the documentation. See [Help APIs and UI tooltips \(page 141\)](#) for details.

Video tutorial on collections

See this [video tutorial on Jekyll.tips](#) for more details on collections.

WebStorm Text Editor

Summary: You can use a variety of text editors when working with a Jekyll project. WebStorm from IntelliJ offers a lot of project-specific features, such as find and replace, that make it ideal for working with tech comm projects.

About text editors and WebStorm

There are a variety of text editors available, but I like WebStorm the best because it groups files into projects, which makes it easy to find all instances of a text string, to do find and replace operations across the project, and more.

If you decide to use WebStorm, here are a few tips on configuring the editor.

Remove unnecessary plugins

By default, WebStorm comes packaged with a lot more functionality than you probably need. You can lighten the editor by removing some of the plugins. Go to **WebStorm > Preferences > Plugins** and clear the check boxes of plugins you don't need.

Set default tab indent to 3 spaces instead of 4

You can set the way the tab works, and whether it uses spaces or a tab character. For details, see [Code Style. JavaScript](#) in WebStorm's help.

On a Mac, go to **WebStorm > Preferences > Editor > Code Style > Other File Types**. Don't select the "Use tab character" check box. Set **4** for the **Tab size** and **Indent** check boxes.

On Windows, go to **File > Settings > Editor > Code Style > Other File Types** to access the same menu.

Add the Markdown Support plugin

Since you'll be writing in Markdown, having color coding and other support for Markdown is important. Install the Markdown Support plugin by going to **WebStorm > Preferences > Plugins** and clicking **Install JetBrains Plugin**. Search for **Markdown Support**. You can also implement the Markdown Navigator plugin.

Enable Soft Wraps (word wrapping)

Most likely you'll want to enable soft wraps, which wraps lines rather than extending them out forever and requiring you to scroll horizontally to see the text. To enable softwrapping, go to **WebStorm > Preferences > Editor > General** and see the Soft Wraps section. Select the **Use soft wraps in editor** check box.

Exclude a directory

When you're searching for content, you don't want to edit any file that appears in the `_site` directory. You can exclude a directory from Webstorm by right-clicking the directory and choosing **Mark Directory As** and then selecting **Excluded**.

Set tabs to 4 spaces

You can set the default number of spaces a tab sets, including whether Webstorm uses a tab character or spaces. You want spaces, and you want to set this to default number of spaces to `4`. Note that this is due to the way Kramdown handles the continuation of lists.

To set the indentation, see the "Tabs and Indents" topic in this [Code Style Javascript](#) topic in Webstorm's help.

Shortcuts

It can help to learn a few key shortcuts:

| Command | Shortcuts |
|---------------------|---|
| Shift + Shift | Allows you to find a file by searching for its name. |
| Shift + Command + F | Find in whole project. (WebStorm uses the term "Find in path".) |
| Shift + Command + R | Replace in whole project. (Again, WebStorm calls it "Replace in path".) |
| Command + F | Find on page |
| Shift + R | Replace on page |

| Command | Shortcuts |
|-----------------------------------|--|
| Right-click >
Add to Favorites | Allows you to add files to a Favorites section, which expands below the list of files in the project pane. |
| Shift + tab | Applies outdenting (opposite of tabbing) |
| Shift + Function
+ F6 | Rename a file |
| Command +
Delete | Delete a file |
| Command + 2 | Show Favorites pane |
| Shift + Option +
F | Add to Favorites |

✔ **Tip:** If these shortcut keys aren't working for you, make sure you have the "Max OS X 10.5+" keymap selected. Go to **WebStorm > Preferences > Keymap** and select it there.

Finding files

When I want to find a file, I browse to the file in the preview site and copy the page name in the URL. Then in Webstorm I press **Shift** twice and paste in the file name. The search feature automatically highlights the file I want, and I press **Enter**.

Identifying changed files

When you have the Git and Github integration, changed files appear in blue. This lets you know what needs to be committed to your repository.

Creating file templates

Rather than insert the frontmatter by hand each time, it's much faster to simply create a Jekyll template. To create a Jekyll template in WebStorm:


1. Right-click a file in the list of project files, and select **New > Edit File Templates**.

If you don't see the Edit File Templates option, you may need to create a file template first. Go to **File > Default Settings > Editor > File and Code Templates**. Create a new file template with an md extension, and then close and restart WebStorm. Then repeat this step and you will see the File Templates option appear in the right context menu.

2. In the upper-left corner of the dialog box that appears, click the + button to create a new template.
3. Name it something like Jekyll page. Insert the frontmatter you want, and save it.

To use the Jekyll template, when you create a new file in your WebStorm project, you can select your Jekyll file template.

Disable pair quotes

By default, each time you type , WebStorm will pair the quote (creating two quotes). You can disable this by going to **WebStorm > Preferences > Editor > Smartkeys**. Clear the **Insert pair quotes** check box.

Atom Text Editor

Summary: Atom is a free text editor that is a favorite tool of many writers because it is free. This page provides some tips for using Atom.

If you haven't downloaded [Atom](#), download and install it. Use this as your editor when working with Jekyll. The syntax highlighting is probably the best among the available editors, as it was designed with Jekyll-authoring in mind. However, if you prefer Sublime Text, WebStorm, or some other editor, you can also use that.

Customize the invisibles and tab spacing in Atom:

1. Go to **Atom > Preferences**.
2. On the **Settings** tab, keep the default options but also select the following:
 - **Show Invisibles**
 - **Soft Wrap**
 - For the **Tab Length**, type **4**.
 - For the **Tab Type**, select **soft**.

Turn off auto-complete:

1. Go to **Atom > Preferences**.
2. Click the **Packages** tab.
3. Search for **autocomplete-plus**.
4. Disable the autocomplete package.

Atom Shortcuts

- **Cmd + T**: Find file
- **Cmd + Shift + F**: Find across project
- **Cmd + Alt + S**: Save all

(For Windows, replace “Cmd” with “Ctrl”.)

Sidebar Navigation

Summary: The sidebar navigation uses a jQuery component called Navgoco. The sidebar is a somewhat complex part of the theme that remembers your current page, highlights the active item, stays in a fixed position on the page, and more. This page explains a bit about how the sidebar was put together.

Navgoco foundation

The sidebar uses the [Navgoco jQuery plugin](#) as its basis. Why not use Bootstrap? Navgoco provides a few features that I couldn't find in Bootstrap:

- Navgoco sets a cookie to remember the user's position in the sidebar. If you refresh the page, the cookie allows the plugin to remember the state.
- Navgoco inserts an `active` class based on the navigation option that's open. This is essential for keeping the accordion open.
- Navgoco includes the expand and collapse features of a sidebar.

In short, the sidebar has some complex logic here. I've integrated Navgoco's features with the sidebar.html and sidebar data files to build the sidebar. It's probably the most impressive part of this theme. (Other themes usually aren't focused on creating hierarchies of pages, but this kind of hierarchy is important in a documentation site.)

Accordion sidebar feature

The sidebar.html file (inside the `_includes` folder) contains the `.navgoco` method called on the `#mysidebar` element.

There are some options to set within the `.navgoco` method. The only noteworthy option is `accordion`. This option makes it so when you expand a section, the other sections collapse. It's a way of keeping your navigation controls condensed.

The value for `accordion` is a Boolean (`true` or `false`). By default, the `accordion` option is set as `true`. If you don't want the accordion, set it to `false`. Note that there's also a block of code near the bottom of sidebar.html that is commented out. Uncomment out that section to have the Collapse all and Expand All buttons appear.

There's a danger with setting the accordion to `false`. If you click Expand All and the sidebar expands beyond the dimensions of the browser, users will be stuck. When that happens, it's hard to collapse it. As a best practice, leave the sidebar's accordion option set to `true`.

Fixed position sidebar

The sidebar has one other feature — this one from Bootstrap. If the user's viewport is tall enough, the sidebar remains fixed on the page. This allows the user to scroll down the page and still keep the sidebar in view.

In the `customscripts.js` file in the `js` folder, there's a function that adds an `affix` class if the height of the browser window is greater than 800 pixels. If the browser's height is less than 800 pixels, the `nav affix` class does not get inserted. As a result, the sidebar can slide up and down as the user scrolls up and down the page.

Depending on your content, you may need to adjust `800` pixel number. If your sidebar is so long that having it in a fixed position makes it so the bottom of the sidebar gets cut off, increase the `800` pixel number here to a higher number.

Opening sidebar links into external pages

In the attributes for each sidebar item, if you use `external_url` instead of `url`, the theme will insert the link into an `a href` element that opens in a blank target.

For example, the `sidebar.html` file contains the following code:

```
{% if folderitem.external_url %}
  <li><a href="{{folderitem.external_url}}" target="_blank" r
el="noopener">{{folderitem.title}}</a></li>
```

You can see that the `external_url` is a condition that applies a different formatting. Although this feature is available, I recommend putting any external navigation links in the top navigation bar instead of the side navigation bar.

Sidebar item highlighting

The `sidebar.html` file inserts an `active` class into the sidebar element when the `url` attribute in the sidebar data file matches the page URL.

For example, the `sidebar.html` file contains the following code:

```
{% elsif page.url == folderitem.url %}  
  <li class="active"><a href="{{folderitem.url | remove:  
  "/" }}">{{folderitem.title}}</a></li>
```

If the `page.url` matches the `subfolderitem.url`, then an `active` class gets applied. If not, the `active` class does not get applied.

The `page.url` in Jekyll is a site-wide variable. If you insert `{{page.url}}` on a page, it will render as follows: `/mydoc_sidebar_navigation.html`. The `url` attribute in the sidebar item must match the page URL in order to get the `active` class applied.

This is why the `url` value in the sidebar data file looks something like this:

```
- title: Understanding how the sidebar works  
  permalink: mydoc_understand_sidebar.html  
  output: web, pdf
```

Note that the url does not include the project folder where the file is stored. This is because the site uses permalinks, which pulls the topics out of subfolders and places them into the root directory when the site builds.

Now the `page.url` and the `item.url` can match and the `active` class can get applied. With the `active` class applied, the sidebar section remains open.

YAML tutorial in the context of Jekyll

Summary: YAML is a format that relies on white spacing to separate out the various elements of content. Jekyll lets you use Liquid with YAML as a way to parse through the data. Storing items for your table of contents is one of the most common uses of YAML with Jekyll.

Overview

One of the most interesting features of Jekyll is the ability to separate out data elements from formatting elements using a combination of YAML and Liquid. This setup is most common when you're trying to create a table of contents.

Not many Jekyll themes actually have a robust table of contents, which is critical when you are creating any kind of documentation or reference material that has a lot of pages.

Here's the basic approach in creating a table of contents. You store your data items in a YAML file using YAML syntax. (I'll go over more about YAML syntax in a later section.) You then create your HTML structure in another file, such as sidebar.html. You might leverage one of the many different table of content frameworks (such as [Navgoco](#)) that have been created for this HTML structure.

Then, using Liquid syntax for loops and conditions, you access all of those values from the data file and splice them into HTML formatting. This will become more clear as we go through some examples.

YAML overview

Rather than just jump into YAML at the most advanced level, I'm going to start from ground zero with an introduction to YAML and how you access basic values in your data files using Jekyll.

Note that you don't actually have to use Jekyll when using YAML. YAML is used in a lot of other systems and is a format completely independent of Jekyll. However, because Jekyll uses Liquid, it gives you a lot of power to parse through your YAML data and make use of it.

YAML itself doesn't do anything on its own — it's just a way of storing your data in a specific structure that other utilities can parse.

YAML basics

You can read about YAML from a lot of different sources. Here are some basic characteristics of YAML:

- YAML (“YAML Ain’t Markup Language”) doesn’t use markup tags. This means you won’t see any kind of angle brackets. It uses white space as a way to form the structure. This makes YAML much more human readable.
- Because YAML does use white space for the structure, YAML is extremely picky about the exactness of spaces. If you have just one extra space somewhere, it can cause the whole file to be invalid.
- For each new level in YAML, you indent two spaces. Each level provides a different access point for the content. You use dot notation to access each new level.
- Because tabs are not universally implemented the same way in editors, a tab might not equate to two spaces. In general, it’s best to manually type two spaces to create a new level in YAML.
- YAML has several types of elements. The most common are mappings and lists. A mapping is simply a key-value pair. A list is a sequence of items. List start with hyphens.
- Items at each level can have various properties. You can create conditions based on the properties.
- You can use “for” loops to iterate through a list.

I realize a lot of this vague and general; however, it will become a lot more clear as we go through some concrete examples.

In the `_data` folder, there’s a file called `samplelist.yml`. All of these examples come from that file.

Example 1: Simple mapping

YAML:

```
name:  
  husband: Tom  
  wife: Shannon
```

Markdown + Liquid:

```
<p>Husband's name: {{site.data.samplelist.name.husband}}</p>
<p>Wife's name: {{site.data.samplelist.name.wife}}</p>
```

Notice that in order to access the data file, you use `site.data.samplelist` where as `samplelist` is the name of the YAML file.

Result:

```
Husband's name: Tom
Wife's name: Shannon
```

Example 2: Line breaks**YAML:**

```
feedback: >
  This is my feedback to you.
  Even if I include linebreaks here,
  all of the linebreaks will be removed when the value is inser
  ted.

block: |
  This pipe does something a little different.
  It preserves the breaks.
  This is really helpful for code samples,
  since you can format the code samples with
  the appropriate
```

Markdown:

```
<p><b>Feedback</b></p>
<p>{{site.data.samplelist.feedback}}</p>

<p><b>Block</b></p>
<p>{{site.data.samplelist.block}}</p>
```

Result:

Feedback

This is my feedback to you. Even if I include linebreaks here, all of the linebreaks will be removed when the value is inserted.

Block

This pipe does something a little different. It preserves the breaks. This is really helpful for code samples, since you can format the code samples with the appropriate white spacing.

The right angle bracket `>` allows you to put the value on the next lines (which must be indented). Even if you create a line break, the output will remove all of those line breaks, creating one paragraph.

The pipe `|` functions like the angle bracket in that it allows you to put the values for the mapping on the next lines (which again must be indented). However, the pipe does preserve all of the line breaks that you use. This makes the pipe method ideal for storing code samples.

Example 3: Simple list

YAML:

```
bikes:
- title: mountain bikes
- title: road bikes
- title: hybrid bikes
```

Markdown + Liquid:

```
<ul>
{% for item in site.data.samplelist.bikes %}
<li>{{item.title}}</li>
{% endfor %}
</ul>
```

Result:

- mountain bikes

- road bikes
- hybrid bikes

Here we use a “for” loop to get each item in the bikes list. By using `.title` we only get the `title` property from each list item.

Example 4: List items

YAML:

```
salesteams:
- title: Regions
  subfolderitems:
    - location: US
    - location: Spain
    - location: France
```

Markdown + Liquid:

```
{% for item in site.data.samplelist.salesteams %}
<h3>{{item.title}}</h3>
<ul>
  {% for entry in item.subfolderitems %}
  <li>{{entry.location}}</li>
  {% endfor %}
</ul>
{% endfor %}
```

Result:

Regions

- US
- Spain
- France

Hopefully you can start to see how to wrap more complex formatting around the YAML content. When you use a “for” loop, you choose the variable of what to call the list items. The variable you choose to use becomes how you access the properties of each list item. In this case, I decided to use the variable `item`. In order to get each property of the list item, I used `item.subfolderitems`.

Each list item starts with the hyphen `-`. You cannot directly access the list item by referring to a mapping. You only loop through the list items. If you wanted to access the list item, you would have to use something like `[1]`, which is how you access the position in an array. You cannot access a list item like you can access a mapping key.

Example 5: Table of contents

YAML:

```
toc:
  - title: Group 1
    subfolderitems:
      - page: Thing 1
      - page: Thing 2
      - page: Thing 3
  - title: Group 2
    subfolderitems:
      - page: Piece 1
      - page: Piece 2
      - page: Piece 3
  - title: Group 3
    subfolderitems:
      - page: Widget 1
      - page: Widget 2 it's
      - page: Widget 3
```

Markdown + Liquid:

```
{% for item in site.data.samplelist.toc %}  
<h3>{{item.title}}</h3>  
<ul>  
  {% for entry in item.subfolderitems %}  
    <li>{{entry.page}}</li>  
  {% endfor %}  
</ul>  
{% endfor %}
```

Result:**Group 1**

- Thing 1
- Thing 2
- Thing 3

Group 2

- Piece 1
- Piece 2
- Piece 3

Group 3

- Widget 1
- Widget 2
- Widget 3

This example is similar to the previous one, but it's more developed as a real table of contents.

Example 6: Variables

YAML:

```
something: &hello Greetings earthling!  
myref: *hello
```

Markdown:

```
{{ site.data.samplelist.myref }}
```

Result:

```
Greetings earthling!
```

This example is notably different. Here I'm showing how to reuse content in YAML file. If you have the same value that you want to repeat in other mappings, you can create a variable using the `&` symbol. Then when you want to refer to that variable's value, you use an asterisk `*` followed by the name of the variable.

In this case the variable is `&hello` and its value is `Greetings earthling!` In order to reuse that same value, you just type `*hello`.

I don't use variables much, but that's not to say they couldn't be highly useful. For example, let's say you put name of the product in parentheses after each title (because you have various products that you're providing documentation for in the same site). You could create a variable for that product name so that if you change how you're referring to it, you wouldn't have to change all instances of it in your YAML file.

Example 7: Positions in lists

YAML:

```
about:  
- zero  
- one  
- two  
- three
```

Markdown:

```
{{ site.data.samplelist.about[0] }}
```

Result:

```
zero
```

You can see that I'm accessing one of the items in the list using `[0]`. This refers to the position in the array where a list item is. Like most programming languages, you start counting at zero, not one.

I wanted to include this example because it points to the challenge in getting a value from a specific list item. You can't just call out a specific item in a list like you can with a mapping. This is why you usually iterate through the list items using a "for" loop.

Example 8: Properties from list items at specific positions

YAML:

```
numbercolors:
- zero:
  properties: red
- one:
  properties: yellow
- two:
  properties: green
- three:
  properties: blue
```

Markdown + Liquid:

```
{{ site.data.samplelist.numbercolors[0].properties }}
```

Result:

```
red
```

This example is similar as before; however, in this case we were getting a specific property from the list item in the zero position.

Example 9: Conditions

YAML:


```
mypages:
- section1: Section 1
  audience: developers
  product: acme
  url: facebook.com
- section2: Section 2
  audience: writers
  product: acme
  url: google.com
- section3: Section 3
  audience: developers
  product: acme
  url: amazon.com
- section4: Section 4
  audience: writers
  product: gizmo
  url: apple.com
- section5: Section 5
  audience: writers
  product: acme
  url: microsoft.com
```

Markdown + Liquid:

```
<ul>
{% for sec in site.data.samplelist.mypages %}
{% if sec.audience == "writers" %}
<li>{{sec.url}}</li>
{% endif %}
{% endfor %}
</ul>
```

Result:

- google.com
- apple.com
- microsoft.com

This example shows how you can use conditions in order to selectively get the YAML content. In your table of contents, you might have a lot of different pages. However, you might only want to get the pages for a particular audience. Conditions lets you get only the items that meet those audience attributes.

Now let's adjust the condition just a little. Let's add a second condition so that the `audience` property has to be `writers` and the `product` property has to be `gizmo`. This is how you would write it:

```
<ul>
{% for sec in site.data.samplelist.mypages %}
{% if sec.audience == "writers" and sec.product == "gizmo" %}
<li>{{sec.url}}</li>
{% endif %}
{% endfor %}
</ul>
```

And here is the result:

- [apple.com](#)

More resources

For more examples and explanations, see this helpful post on [tournemille.com](#): [How to create data-driven navigation in Jekyll](#) .

Tags

Summary: Tags provide another means of navigation for your content. Unlike the table of contents, tags can show the content in a variety of arrangements and groupings. Implementing tags in this Jekyll theme is somewhat of a manual process.

Add a tag to a page

You can add tags to pages by adding `tags` in the frontmatter with values inside brackets, like this:

```
---
title: 5.0 Release Notes
permalink: release_notes_5_0.html
tags: [formatting, single_sourcing]
---
```

Tags overview

Note: With posts, tags have a namespace that you can access with `posts.tags.tagname`, where `tagname` is the name of the tag. You can then list all posts in that tag namespace. But pages don't off this same tag namespace, so you could actually use another key instead of `tags`. Nevertheless, I'm using the same tags approach for posts as with pages.

To prevent tags from getting out of control and inconsistent, first make sure the tag appears in the `_data/tags.yml` file. If it's not there, the tag you add to a page won't be read. I added this check just to make sure I'm using the same tags consistently and not adding new tags that don't have tag archive pages.

Note: In contrast to WordPress, with Jekyll to get tags on pages you have to build out the functionality for tags so that clicking a tag name shows you all pages with that tag. Tags in Jekyll are much more manual.

Additionally, you must create a tag archive page similar to the other pages named `tag_{tagname}.html` folder. This theme doesn't auto-create tag archive pages.

For simplicity, make all your tags single words (connect them with hyphens if necessary).

Setting up tags

Tags have a few components.

1. In the `_data/tags.yml` file, add the tag names you want to allow. For example:

```
allowed-tags:
  - getting_started
  - overview
  - formatting
  - publishing
  - single_sourcing
  - special_layouts
  - content_types
```

2. Create a tag archive file for each tag in your `tags_doc.yml` list. Name the file following the same pattern in the `tags` folder, like this:
`tag_collaboration.html`.

Each tag archive file needs only this:

```
---
title: "Collaboration pages"
tagName: collaboration
search: exclude
permalink: tag_collaboration.html
sidebar: mydoc_sidebar
---
{% include taglogic.html %}
```

❗ Note: In the `_includes/mydoc` folder, there's a `taglogic.html` file. This file (included in each tag archive file) has common logic for getting the tags and listing out the pages containing the tag in a table with summaries or truncated excerpts. You don't have to do anything with the file — just leave it there because the tag archive pages reference it.

3. Change the title, tagName, and permalink values to be specific to the tag name you just created.

By default, the `_layouts/page.html` file will look for any tags on a page and insert them at the bottom of the page using this code:

```
<div class="tags">
{% if page.tags != null %}
<b>Tags: </b>
{% assign projectTags = site.data.tags.allowed-tags %}
{% for tag in page.tags %}
{% if projectTags contains tag %}
<a href="{{ "tag_" | append: tag | append: ".html" }}" class="btn btn-default navbar-btn cursorNorm" role="button">{{page.tagName}}
{{tag}}</a>
{% endif %}
{% endfor %}
{% endif %}
</div>
```

Because this code appears on the `_layouts/page.html` file by default, you don't need to do anything in your page to get the tags to appear. However, if you want to alter the placement or change the button color, you can do so within the `_includes/taglogic.html` file.

You can change the button color by changing the class on the button from `btn-info` to one of the other button classes bootstrap provides. See [Labels \(page 104\)](#) for more options on button class names.

Retrieving pages for a specific tag

If you want to retrieve pages outside of a particular `tag_archive` page, you could use this code:

```
Getting started pages:
<ul>
{% for page in site.pages %}
{% for tag in page.tags %}
{% if tag == "getting_started" %}
<li><a href="{{page.url | remove: "/" }}">{{page.title}}</a></li>
{% endif %}
{% endfor %}
{% endfor %}
</ul>
```

Here's how that code renders:

Getting started pages:

- [Blain's World @ ACC \(page 3\)](#)
- [About the theme's author \(page 10\)](#)
- [About Ruby, Gems, Bundler, and other prerequisites \(page 16\)](#)
- [Install Jekyll on Mac \(page 24\)](#)
- [Pages \(page 32\)](#)
- [Posts \(page 39\)](#)
- [Release notes 5.0 \(page 14\)](#)
- [Release notes 6.0 \(page 12\)](#)
- [Sidebar Navigation \(page 59\)](#)
- [Support \(page 11\)](#)
- [Supported features \(page 5\)](#)

If you want to sort the pages alphabetically, you have to apply a `sort` filter:

```
Getting started pages:
<ul>
{% assign sorted_pages = site.pages | sort: 'title' %}
{% for page in sorted_pages %}
{% for tag in page.tags %}
{% if tag == "getting_started" %}
<li><a href="{{page.url | remove: '/' }}">{{page.title}}</a></li>
{% endif %}
{% endfor %}
{% endfor %}
</ul>
```

Here's how that code renders:

Getting started pages:

- [About Ruby, Gems, Bundler, and other prerequisites \(page 16\)](#)
- [About the theme's author \(page 10\)](#)
- [Blain's World @ ACC \(page 3\)](#)
- [Install Jekyll on Mac \(page 24\)](#)
- [Pages \(page 32\)](#)
- [Posts \(page 39\)](#)
- [Release notes 5.0 \(page 14\)](#)
- [Release notes 6.0 \(page 12\)](#)
- [Sidebar Navigation \(page 59\)](#)
- [Support \(page 11\)](#)
- [Supported features \(page 5\)](#)

Efficiency

Although the tag approach here uses `for` loops, these are somewhat inefficient on a large site. Most of my tech doc projects don't have hundreds of pages (like my blog does). If your project does have hundreds of pages, this `for` loop approach with tags is going to slow down your build times.

Without the ability to access pages inside a universal namespace with the page type, there aren't many workarounds here for faster looping.

With posts (instead of pages), since you can access just the posts inside `posts.tag.tagname`, you can be a lot more efficient with the looping.

Still, if the build times are getting long (e.g., 1 or 2 minutes per build), look into reducing the number of `for` loops on your site.

Empty tags?

If your page shows “tags:” at the bottom without any value, it could mean a couple of things:

- You’re using a tag that isn’t specified in your allowed tags list in your tags.yml file.
- You have an empty `tags: []` property in your frontmatter.

If you don’t want tags to appear at all on your page, remove the tags property from your frontmatter.

Remembering the right tags

Since you may have many tags and find it difficult to remember what tags are allowed, I recommend creating a template that prepopulates all your frontmatter with all possible tags. Then just remove the tags that don’t apply.

See [WebStorm Text Editor \(page 54\)](#) for tips on creating file templates in WebStorm.

Series

Summary: You can automatically link together topics belonging to the same series. This helps users know the context within a particular process.

Using series for pages

You create a series by looking for all pages within a tag namespace that contain certain frontmatter. Here's a [demo \(page 0\)](#).

1. Create the series button

First create an include that contains your series button:

```
<div class="seriesContext">
  <div class="btn-group">
    <button type="button" data-toggle="dropdown" class="btn btn-primary dropdown-toggle">Series Demo <span class="caret"></span></button>
    <ol class="dropdown-menu">
      {% assign pages = site.pages | sort:"weight" %}
      {% for p in pages %}
      {% if p.series == "ACME series" %}
      {% if p.url == page.url %}
      <li class="active"> → {{p.weight}}. {{p.title}}</li>
      {% else %}
      <li>
        <a href="{{p.url | remove: "/"}}">{{p.weight}}. {{p.title}}</a>
      </li>
      {% endif %}
      {% endif %}
      {% endfor %}
    </ol>
  </div>
</div>
```

Change “ACME series” to the name of your series.

Save this in your `_includes/custom` folder as something like `series_acme.html`.

⚠ Warning: With pages, there isn't a universal namespace created from tags or categories like there is with Jekyll posts. As a result, you have to loop through all pages. If you have a lot of pages in your site (e.g., 1,000+), then this looping will create a slow build time. If this is the case, you will need to rethink the approach to looping here.

2. Create the “next” include

Now create another include for the Next button at the bottom of the page. Copy the following code, changing the series name to your series' name:

```
<p>{% assign series_pages = site.tags.series_acme %}
  {% for p in pages %}
    {% if p.series == "ACME series" %}
      {% assign nextTopic = page.weight | plus: "1" %}
      {% if p.weight == nextTopic %}
        <a href="{{p.url}}"><button type="button" class="btn btn-primary">Next: {{p.weight}} {{p.title}}</button></a>
      {% endif %}
    {% endif %}
  {% endfor %}
</p>
```

Change “acme” to the name of your series.

Save this in your `_includes/custom/mydoc` folder as `series_acme_next.html`.

3. Add the correct frontmatter to each of your series pages

Now add the following frontmatter to each page in the series:

```
series: "ACME series"
weight: 1.0
```

With weights, Jekyll will treat 10 as coming after 1. If you have more than 10 items, consider changing `plus: "1.0"` to `plus: "0.1"`.

Additionally, if your page names are prefaced with numbers, such as “1. Download the code,” then the `{{p.weight}}` will create a duplicate number. In that case, just remove the `{{p.weight}}` from both code samples here.

4. Add links to the series button and next button on each page.

On each series page, add a link to the series button at the top and a link to the next button at the bottom.

```
<!-- your frontmatter goes here -->

{% include custom/series_acme.html %}

<!-- your page content goes here ... -->

{% include custom/series_acme_next.html %}
```

Changing the series drop-down color

The Bootstrap menu uses the `primary` class for styling. If you change this class in your theme, the Bootstrap menu should automatically change color as well. You can also just use another Bootstrap class in your button code. Instead of `btn-primary`, use `btn-info` or `btn-warning`. See [Labels \(page 104\)](#) for more Bootstrap button classes.

Using a collection with your series

Instead of copying and pasting the button includes on each of your series, you could also create a collection and define a layout for the collection that has the include code. For more information on creating collections, see [Collections \(page 52\)](#) for more details.

Tooltips

Summary: You can add tooltips to any word, such as an acronym or specialized term. Tooltips work well for glossary definitions, because you don't have to keep repeating the definition, nor do you assume the reader already knows the word's meaning.

Creating tooltips

Because this theme is built on Bootstrap, you can simply use a specific attribute on an element to insert a tooltip.

Suppose you have a glossary.yml file inside your `_data` folder. You could pull in that glossary definition like this:

```
<a href="#" data-toggle="tooltip" data-original-title="{{site.data.glossary.jekyll_platform}}">Jekyll</a> is my favorite tool for building websites.
```

This renders to the following:

Jekyll is my favorite tool for building websites.

Alerts

Summary: You can insert notes, tips, warnings, and important alerts in your content. These notes make use of Bootstrap styling and are available through data references such as `site.data.alerts.note`.

About alerts

Alerts are little warnings, info, or other messages that you have called out in special formatting. In order to use these alerts or callouts, reference the appropriate value stored in the `alerts.yml` file as described in the following sections.

Alerts

Similar to [inserting images \(page 98\)](#), you insert alerts through various includes that have been developed. These includes provide templates through which you pass parameters to easily populate the right HTML code.

```
{% include note.html content="This is my note. All the content  
I type here is treated as a single paragraph." %}
```

Here's the result:

❗ Note: This is my note. All the content I type here is treated as a single paragraph.

With alerts, there's just one include property:

Property	description
content	The content for the alert.


```
def foo(x):  
    return x+1
```

The same Bootstrap code from the alert is stored in yaml files inside the `_data` folder. (This was how I previously implemented this code, but since this method was prone to error and didn't trigger any build warnings or failures when incorrectly coded, I changed the approach to use includes instead.)

Types of alerts available

There are four types of alerts you can leverage:

- `note.html`
- `tip.html`
- `warning.html`
- `important.html`

They function the same except they have a different color, icon, and alert word. You include the different types by selecting the include template you want. Here are samples of each alert:

📌 Note: This is my note.

✅ Tip: This is my tip.

⚠ Warning: This is my warning.

⚠ Important: This is my important info.

These alerts leverage includes stored in the `_include` folder. The `content` option is a parameter that you pass to the include. In the include, the parameter is passed like this:

```
<div markdown="span" class="alert alert-info" role="alert"><i class="fa fa-info-circle"></i> <b>Note:</b> {{include.content}}</div>
```

The content in `content="This is my note."` gets inserted into the `{{include.content}}` part of the template. You can follow this same pattern to build additional includes. See this [Jekyll screencast on includes](#) or [this screencast](#) for more information.

Callouts

There's another type of callout available called callouts. This format is typically used for longer callout that spans more than one or two paragraphs, but really it's just a stylistic preference whether to use an alert or callout.

Here's the syntax for a callout:

```
{% include callout.html content="This is my callout. It has a border on the left whose color you define by passing a type parameter. I typically use this style of callout when I have more information that I want to share, often spanning multiple paragraphs. " type="primary" %}
```

Here's the result:

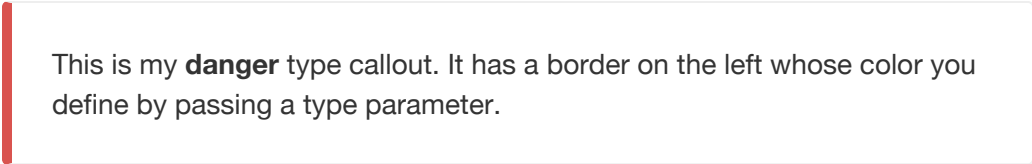
This is my callout. It has a border on the left whose color you define by passing a type parameter. I typically use this style of callout when I have more information that I want to share, often spanning multiple paragraphs.

The available properties for callouts are as follows:

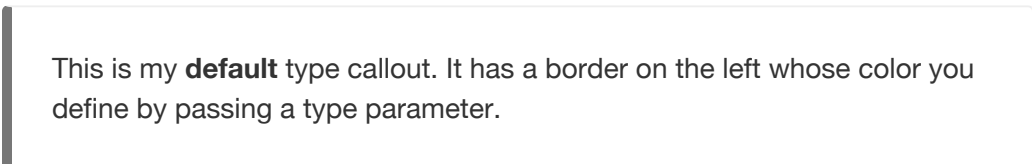
Property	description
content	The content for the callout.
type	The style for the callout. Options are <code>danger</code> , <code>default</code> , <code>primary</code> , <code>success</code> , <code>info</code> , and <code>warning</code> .

The types just define the color of the left border. Each of these callout types get inserted as a class name in the callout template. These class names correspond with styles in Bootstrap. These classes are common Bootstrap class names whose style attributes differ depending on your Bootstrap theme and style definitions.

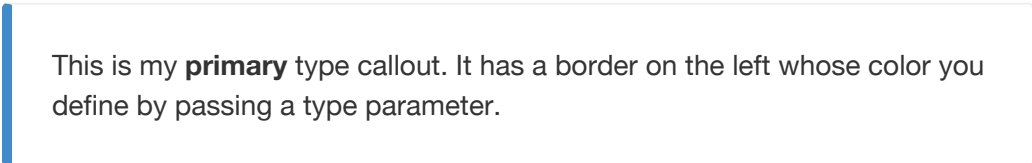
Here's an example of each different type of callout:



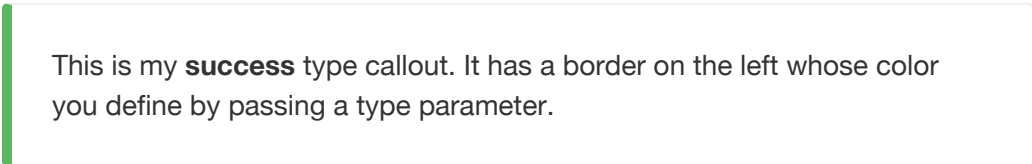
This is my **danger** type callout. It has a border on the left whose color you define by passing a type parameter.



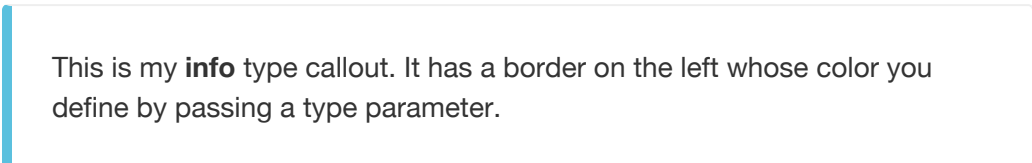
This is my **default** type callout. It has a border on the left whose color you define by passing a type parameter.



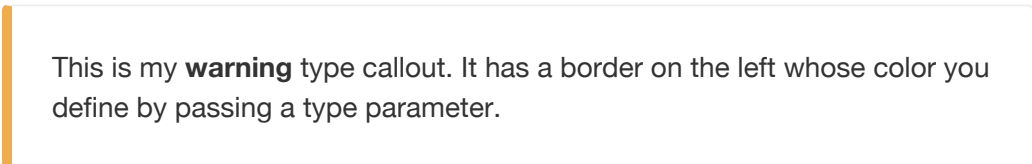
This is my **primary** type callout. It has a border on the left whose color you define by passing a type parameter.



This is my **success** type callout. It has a border on the left whose color you define by passing a type parameter.



This is my **info** type callout. It has a border on the left whose color you define by passing a type parameter.



This is my **warning** type callout. It has a border on the left whose color you define by passing a type parameter.

Now that in contrast to alerts, callouts don't include the alert word (note, tip, warning, or important). You have to manually include it inside **content** if you want it.

To include paragraph breaks, use `

` inside the callout:

```
{% include callout.html content="**Important information**: This is my callout. It has a border on the left whose color you define by passing a type parameter. I typically use this style of callout when I have more information that I want to share, often spanning multiple paragraphs. <br/><br/>Here I am starting a new paragraph, because I have lots of information to share. You may wonder why I'm using line breaks instead of paragraph tags. This is because Kramdown processes the Markdown here as a span rather than a div (for whatever reason). Be grateful that you can be using Markdown at all inside of HTML. That's usually not allowed in Markdown syntax, but it's allowed here." type="primary" %}
```

Here's the result:

Important information: This is my callout. It has a border on the left whose color you define by passing a type parameter. I typically use this style of callout when I have more information that I want to share, often spanning multiple paragraphs.

Here I am starting a new paragraph, because I have lots of information to share. You may wonder why I'm using line breaks instead of paragraph tags. This is because Kramdown processes the Markdown here as a span rather than a div (for whatever reason). Be grateful that you can be using Markdown at all inside of HTML. That's usually not allowed in Markdown syntax, but it's allowed here.

Use Liquid variables inside parameters with includes

Suppose you have a product name or some other property that you're storing as a variable in your configuration file (`_config.yml`), and you want to use this variable in the `content` parameter for your alert or callout. You will get an error if you use Liquid syntax inside a include parameter. For example, this syntax will produce an error:

```
{% include note.html content="The {{site.company}} is pleased to announce an upcoming release." %}
```

The error will say something like this:

```
Liquid Exception: Invalid syntax for include tag. File contains invalid characters or sequences: ... Valid syntax: {% include file.ext param='value' param2='value' %}
```

To use variables in your include parameters, you must use the “variable parameter” approach. First you use a `capture` tag to capture some content. Then you reference this captured tag in your include. Here’s an example.

In my site configuration file (`_config.yml`), I have a property called `company_name`.

```
company_name: Your company
```

I want to use this variable in my note include.

First, before the note I capture the content for my note’s include like this:

```
{% capture company_note %}The {{site.company_name}} company is pleased to announce an upcoming release.{% endcapture %}
```

Now reference the `company_note` in your `include` parameter like this:

```
{% include note.html content=company_note %}
```

Here’s the result:

Note: The Blain’s World is pleased to announce an upcoming release.

Note the omission of quotation marks with variable parameters.

Also note that instead of storing the variable in your site’s configuration file, you could also put the variable in your page’s frontmatter. Then instead of using `{{site.company_name}}` you would use `{{page.company_name}}`.

Markdown inside of callouts and alerts

You can use Markdown inside of callouts and alerts, even though this content actually gets inserted inside of HTML in the include. This is one of the advantages of kramdown Markdown. The include template has an attribute of `markdown="span"` that allows for the processor to parse Markdown inside of HTML.

Validity checking

If you have some of the syntax wrong with an alert or callout, you'll see an error when Jekyll tries to build your site. The error may look like this:

```
Liquid Exception: Invalid syntax for include tag: content="This is my **info** type callout. It has a border on the left whose color you define by passing a type parameter. type="info" Valid syntax: {% include file.ext param='value' param2='value' %} in mydoc/mydoc_alerts.md
```

These errors are a good thing, because it lets you know there's an error in your syntax. Without the errors, you may not realize that you coded something incorrectly until you see the lack of alert or callout styling in your output.

In this case, the quotation marks aren't set correctly. I forgot the closing quotation mark for the content parameter include.

Blast a warning to users on every page

If you want to blast a warning to users on every page, add the alert or callout to the `_layouts/page.html` page right below the frontmatter. Every page using the page layout (all, by default) will show this message.

Icons

Summary: You can integrate font icons through the Font Awesome and Glyphical Halflings libraries. These libraries allow you to embed icons through their libraries delivered as a link reference. You don't need any image libraries downloaded in your project.

Font icon options

The theme has two font icon sets integrated: Font Awesome and Glyphicons Halflings. The latter is part of Bootstrap, while the former is independent. Font icons allow you to insert icons drawn as vectors from a CDN (so you don't have any local images on your own site).

External icons

When you link to an external site, like [Jekyll](#), an icon appears after the link. If you want to remove this icon, comment out this style in `css/customstyles.css`.

```
/* this part adds an icon after external links, using FontAwesome */  
a[href^="http://"]:after, a[href^="https://"]:after {  
  content: "\f08e";  
  font-family: FontAwesome;  
  font-weight: normal;  
  font-style: normal;  
  display: inline-block;  
  text-decoration: none;  
  padding-left: 3px;  
}
```

See Font Awesome icons available

Go to the [Font Awesome library](#) to see the available icons.

The Font Awesome icons allow you to adjust their size by simply adding `fa-2x`, `fa-3x` and so forth as a class to the icon to adjust their size to two times or three times the original size. As vector icons, they scale crisply at any size.

Here's an example of how to scale up a camera icon:

```

<i class="fa fa-camera-retro"></i> normal size (1x)
<i class="fa fa-camera-retro fa-lg"></i> fa-lg
<i class="fa fa-camera-retro fa-2x"></i> fa-2x
<i class="fa fa-camera-retro fa-3x"></i> fa-3x
<i class="fa fa-camera-retro fa-4x"></i> fa-4x
<i class="fa fa-camera-retro fa-5x"></i> fa-5x

```

Here's what they render to:



With Font Awesome, you always use the `i` tag with the appropriate class. You also implement `fa` as a base class first. You can use font awesome icons inside other elements. Here I'm using a Font Awesome class inside a Bootstrap alert:

```

<div class="alert alert-danger" role="alert"><i class="fa fa-exclamation-circle"></i> <b>Warning: </b>This is a special warning message.

```

Here's the result:

! This is a special warning message.

The notes, tips, warnings, etc., are pre-coded with Font Awesome and stored in the alerts.yml file. That file includes the following:

```

tip: '<div class="alert alert-success" role="alert"><i class="fa fa-check-square-o"></i> <b>Tip: </b>'
note: '<div class="alert alert-info" role="alert"><i class="fa fa-info-circle"></i> <b>Note: </b>'
important: '<div class="alert alert-warning" role="alert"><i class="fa fa-warning"></i> <b>Important: </b>'
warning: '<div class="alert alert-danger" role="alert"><i class="fa fa-exclamation-circle"></i> <b>Warning: </b>'
end: '</div>'

callout_danger: '<div class="bs-callout bs-callout-danger">'
callout_default: '<div class="bs-callout bs-callout-default">'
callout_primary: '<div class="bs-callout bs-callout-primary">'
callout_success: '<div class="bs-callout bs-callout-success">'
callout_info: '<div class="bs-callout bs-callout-info">'
callout_warning: '<div class="bs-callout bs-callout-warning">'

hr_faded: '<hr class="faded"/>'
hr_shaded: '<hr class="shaded"/>'

```

This means you can insert a tip, note, warning, or important alert simply by using these tags.

```
{% include note.html content="Add your note here." %}
```

```
{% include tip.html content="Add your tip here." %}
```

```
{% include important.html content="Add your important info here." %}
```

```
{% include warning.html content="Add your warning here." %}
```

Here's the result:

Note: Add your note here.

Tip: Here's my tip.

⚠ Important: This information is very important.

⚠ Warning: If you overlook this, you may die.

The color scheme is the default colors from Bootstrap. You can modify the icons or colors as needed.

Creating your own combinations

You can innovate with your own combinations. Here's a similar approach with a file download icon:

```
<div class="alert alert-success" role="alert"><i class="fa fa-download fa-lg"></i> This is a special tip about some file to download....</div>
```

And the result:

 This is a special tip about some file to download....

Grab the right class name from the [Font Awesome library](#) and then implement it by following the pattern shown previously.

If you want to make your fonts even larger than the 5x style, add a custom style to your stylesheet like this:

```
.fa-10x{font-size:1700%;}
```

Then any element with the attribute `fa-10x` will be enlarged 1700%.

Glyphicon icons available

Glyphicons work similarly to Font Awesome. Go to the [Glyphicons library](#) to see the icons available.

Although the Glyphicon Halflings library doesn't provide the scalable classes like Font Awesome, there's a [StackOverflow trick](#) to make the icons behave in a similar way. This theme's stylesheet (customstyles.css) includes the following to the stylesheet:

```
.gi-2x{font-size: 2em;}  
.gi-3x{font-size: 3em;}  
.gi-4x{font-size: 4em;}  
.gi-5x{font-size: 5em;}
```

Now you just add `gi-5x` or whatever to change the size of the font icon:

```
<span class="glyphicon glyphicon-globe gi-5x"></span>
```

And here's the result:



Glyphicons use the `span` element instead of `i` to attach their classes.

Here's another example:

```
<span class="glyphicon glyphicon-download"></span>
```




And magnified:

```
<span class="glyphicon glyphicon-download gi-3x"></span>
```



You can also put glyphicons inside other elements:


```
<div class="alert alert-danger" role="alert">
  <span class="glyphicon glyphicon-exclamation-sign" aria-hidden="true"></span>
  <b>Error:</b> Enter a valid email address
</div>
```

 **Error:** Enter a valid email address

Callouts

The previously shown alerts might be fine for short messages, but with longer notes, the solid color takes up a bit of space. In this theme, you also have the option of using callouts, which are pretty common in Bootstrap's documentation but surprisingly not offered as an explicit element. Their styles have been copied into this theme, in a way similar to the alerts:

```
<div class="bs-callout bs-callout-info">
  This is a special info message. This is a special info message. This is a special info message. This is a special info message. This is a special info message. This is a special info message. This is a special info message. This is a special info message. </div>
```

 This is a special info message. This is a special info message. This is a special info message. This is a special info message. This is a special info message. This is a special info message. This is a special info message. This is a special info message.

And here's the shortcode:

```
{{site.data.alerts.callout_info}}This is a special callout information message.{{site.data.alerts.end}}
```

Here's the result:



This is a special callout information message.

You can use any of the following:

```
{{site.data.alerts.callout_default}}  
{{site.data.alerts.callout_primary}}  
{{site.data.alerts.callout_success}}  
{{site.data.alerts.callout_info}}  
{{site.data.alerts.callout_warning}}
```

The only difference is the color of the left bar.

Callouts are explained in a bit more detail in [Alerts \(page 83\)](#).

Images

Summary: Store images in the images folder and use the image.html include to insert images. This include has several options, including figcaptions, that extract the content from the formatting.

Image Include Template

Instead of using Markdown or HTML syntax directly in your page for images, the syntax for images has been extracted out into an image include that allows you to pass the parameters you need. Include the image.html like this:

```
{% include image.html file="jekyll.png" url="http://jekyllrb.com" alt="Jekyll" caption="This is a sample caption" %}
```

The available include properties are as follows:

Property	description
file	The name of the file. Store it in the /images folder. If you want to organize your images in subfolders, reference the subfolder path here, like this: <code>mysubfolder/jekyllrb.png</code>
url	Whether to link the image to a URL
alt	Alternative image text for accessibility and SEO
caption	A caption for the image
max-width	a maximum width for the image (in pixels). Just specify the number, not px.

The properties of the include get populated into the image.html template.

Here's the result:



This is a sample caption

Inline image includes

For inline images, such as with a button that you want to appear inline with text, use the `inline_image.html` include, like this:

```
Click the Android SDK Manager button {% include inline_image.html  
file="androidsdkmanagericon.png" alt="SDK button" %}
```

Click the **Android SDK Manager** button 

The `inline_image.html` include properties are as follows:

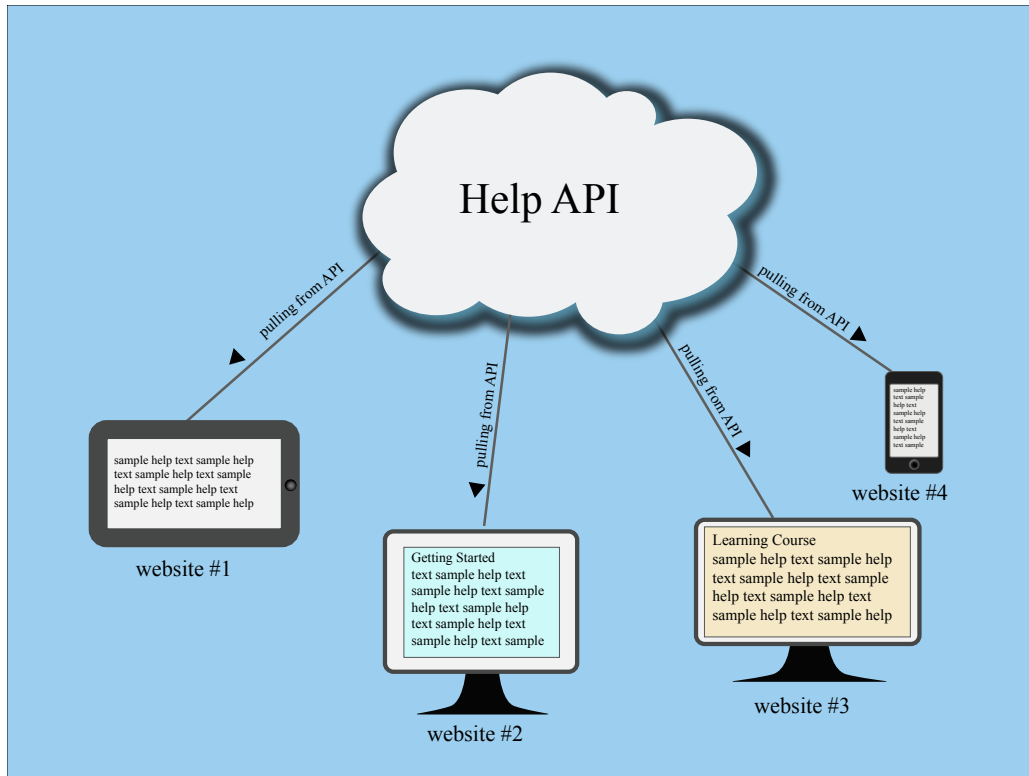
Property	description
file	The name of the file
type	The type of file (png, svg, and so on)
alt	Alternative image text for accessibility and SEO

SVG Images

You can also embed SVG graphics. If you use SVG, you need to use the HTML syntax so that you can define a width/container for the graphic. Here's a sample embed:

```
{% include image.html file="helpapi.svg" url="http://idratherbe  
writing.com/documentation-theme-jekyll/mydoc_help_api/" alt="Bu  
ilding a Help API" caption="A help API provides a JSON file at  
a web URL with content that can be pulled into different target  
s" max-width="600" %}
```

Here's the result:



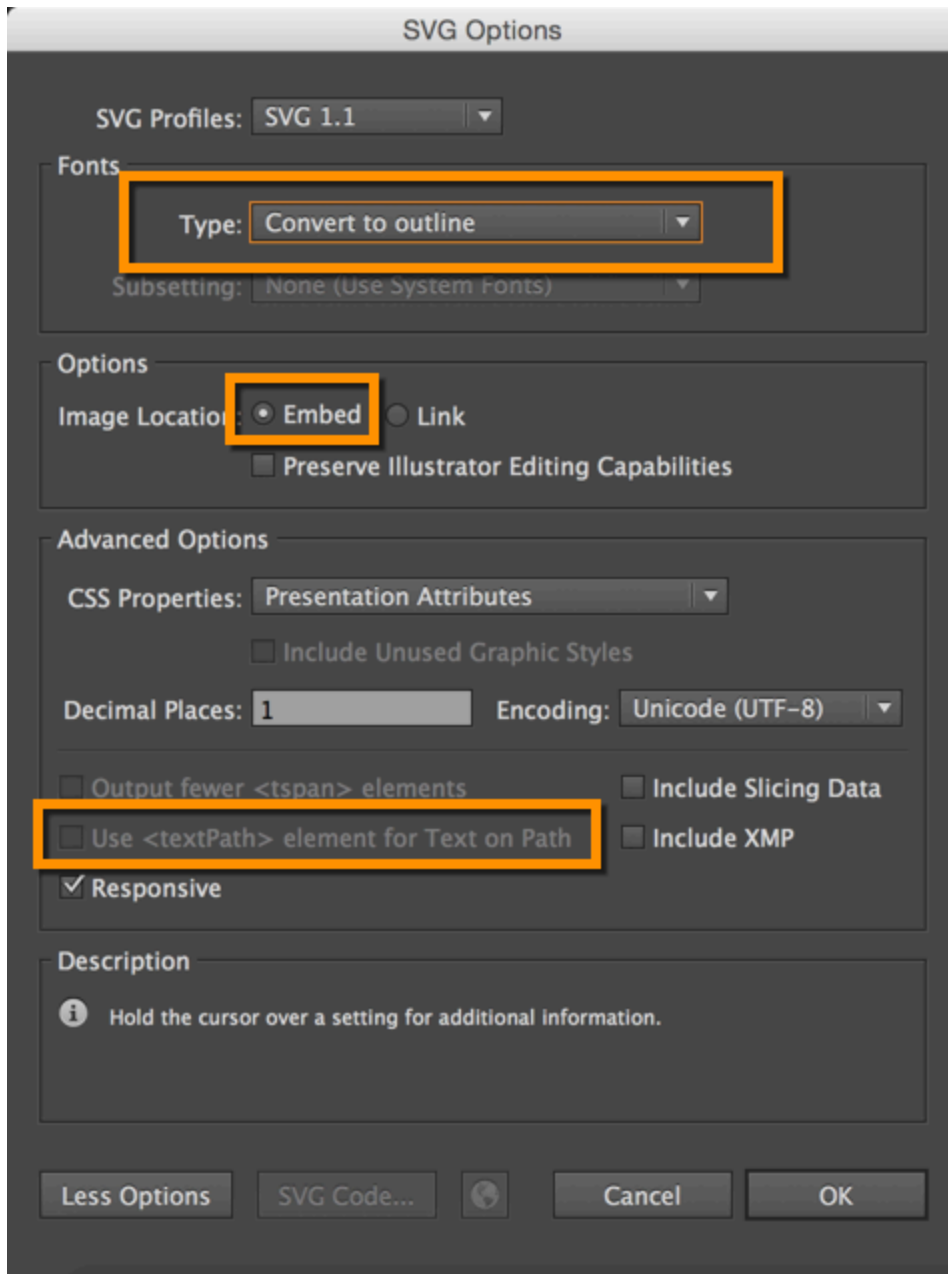
A help API provides a JSON file at a web URL with content that can be pulled into different targets

The stylesheet even handles SVG display in IE 9 and earlier through the following style (based on this [gist](#)):

```
/*
 * Let's target IE to respect aspect ratios and sizes for img tags containing SVG files
 *
 * [1] IE9
 * [2] IE10+
 */
/* 1 */
.ie9 img[src$=".svg"] {
    width: 100%;
}
/* 2 */
@media screen and (-ms-high-contrast: active), (-ms-high-contrast: none) {
    img[src$=".svg"] {
        width: 100%;
    }
}
```

Also, if you're working with SVG graphics, note that Firefox does not support SVG fonts. In Illustrator, when you do a Save As with your AI file and choose SVG, to preserve your fonts, in the Font section, select "Convert to outline" as the Type (don't choose SVG in the Font section).

Also, remove the check box for "Use textpath element for text on a path". And select "Embed" rather than "Link." The following screenshot shows the settings I use. Your graphics will look great in Firefox.



Essential options for SVG with Illustrator

Code samples

Summary: You can use fenced code blocks with the language specified after the first set of backtick fences.

Code Samples

Use fenced code blocks with the language specified, like this:

```
```js
console.log('hello');
```
```

Result:

```
console.log('hello');
```

For the list of supported languages you can use (similar to `js` for JavaScript), see [Supported languages](#).

Labels

Summary: Labels are just a simple Bootstrap component that you can include in your pages as needed. They represent one of many Bootstrap options you can include in your theme.

About labels

Labels might come in handy for adding button-like tags next to elements, such as POST, DELETE, UPDATE methods for endpoints. You can use any classes from Bootstrap in your content.

```
<span class="label label-default">Default</span>
<span class="label label-primary">Primary</span>
<span class="label label-success">Success</span>
<span class="label label-info">Info</span>
<span class="label label-warning">Warning</span>
<span class="label label-danger">Danger</span>
```

Default Primary Success Info Warning Danger

You can have a label appear within a heading simply by including the span tag in the heading. However, you can't mix Markdown syntax with HTML, so you'd have to hard-code the heading ID for the auto-TOC to work.

Links

Summary: When creating links, you can use standard HTML or Markdown formatting. However, you can also implement an automated approach to linking that makes linking much less error-prone (meaning less chances of broken links in your output) and requiring less effort.

Create an external link

When linking to an external site, use Markdown formatting because it's simplest:

```
[Google] (http://google.com)
```

Linking to internal pages

When linking to internal pages, you can manually link to the pages like this:

```
[Icons] (mydoc_icons.html)
```

If you change the file name, you'll have to update all of your links.

Navtabs

Summary: Navtabs provide a tab-based navigation directly in your content, allowing users to click from tab to tab to see different panels of content. Navtabs are especially helpful for showing code samples for different programming languages. The only downside to using navtabs is that you must use HTML instead of Markdown.

Common uses

Navtabs are particularly useful for scenarios where you want to show a variety of options, such as code samples for Java, .NET, or PHP, on the same page.

While you could resort to single-source publishing to provide different outputs for each unique programming language or role, you could also use navtabs to allow users to select the content you want.

Navtabs are better for SEO since you avoid duplicate content and drive users to the same page.

Navtabs demo

The following is a demo of a navtab. Refresh your page to see the tab you selected remain active.

[Profile](#)[About](#)[Match](#)

Profile

Praesent sit amet fermentum leo. Aliquam feugiat,

1. nibh in u ltrices mattis
2. felis ipsum venenatis metus, vel vehicula libero mauris a enim. Sed placerat est ac lectus vestibulum tempor.
 - Quisque ut condimentum massa.
 - ut condimentum massa.

Proin venenatis leo id urna cursus blandit. Vivamus sit amet hendrerit metus.

Code

Here's the code for the above (with the filler text abbreviated):

```
<ul id="profileTabs" class="nav nav-tabs">
  <li class="active"><a href="#profile" data-toggle="tab">Pro
file</a></li>
  <li><a href="#about" data-toggle="tab">About</a></li>
  <li><a href="#match" data-toggle="tab">Match</a></li>
</ul>
<div class="tab-content">
<div role="tabpanel" class="tab-pane active" id="profile">
  <h2>Profile</h2>
<p>Praesent sit amet fermentum leo....</p>
</div>

<div role="tabpanel" class="tab-pane" id="about">
  <h2>About</h2>
  <p>Lorem ipsum ...</p></div>

<div role="tabpanel" class="tab-pane" id="match">
  <h2>Match</h2>
  <p>Vel vehicula ....</p>
</div>
</div>
```

Design constraints

Bootstrap automatically clears any floats after the navtab. Make sure you aren't trying to float any element to the right of your navtabs, or there will be some awkward space in your layout.

Appearance in the mini-TOC

If you put a heading in the navtab content, that heading will appear in the mini-TOC as long as the heading tag has an ID. If you don't want the headings for each navtab section to appear in the mini-TOC, omit the ID attribute from the heading tag. Without this ID attribute in the heading, the mini-TOC won't insert the heading title into the mini-TOC.

Must use HTML

You must use HTML within the navtab content because each navtab section is surrounded with HTML, and you can't use Markdown inside of HTML.

Match up ID tags

Each tab's `href` attribute must match the `id` attribute of the tab content's `div` section. So if your tab has `href="#acme"`, then you add `acme` as the ID attribute in `<div role="tabpanel" class="tab-pane" id="acme">`.

Set an active tab

One of the tabs needs to be set as active, depending on what tab you want to be open by default (usually the first one).

```
<div role="tabpanel" class="tab-pane active" id="acme">
```

Sets a cookie

The navtabs are part of Bootstrap, but this theme sets a cookie to remember the last tab's state. The `js/customscripts.js` file has a long chunk of JavaScript that sets the cookie. The JavaScript comes from [this StackOverflow thread](#).

By setting a cookie, if the user refreshes the page, the active tab is the tab the user last selected (rather than defaulting to the default active tab).

Functionality to implement

One piece of functionality I'd like to implement is the ability to set site-wide nav tab options. For example, if the user always chooses PHP instead of Java in the code samples, it would be great to set this option site-wide by default. However, this functionality isn't yet coded.

Tables

Summary: You can format tables using either multimarkdown syntax or HTML. You can also use jQuery datatables (a plugin) if you need more robust tables.

Multimarkdown Tables

You can use Multimarkdown syntax for tables. The following shows a sample:

```
| Priority apples | Second priority | Third priority |  
|-----|-----|-----|  
| ambrosia | gala | red delicious |  
| pink lady | jazz | macintosh |  
| honeycrisp | granny smith | fuji |
```

Result:

Priority apples	Second priority	Third priority
ambrosia	gala	red delicious
pink lady	jazz	macintosh
honeycrisp	granny smith	fuji

Note: You can't use block level tags (paragraphs or lists) inside Markdown tables, so if you need separate paragraphs inside a cell, use `

`.

HTML Tables

If you need a more sophisticated table syntax, use HTML syntax for the table. Although you're using HTML, you can use Markdown inside the table cells by adding `markdown="span"` as an attribute for the `td` tag, as shown in the following table. You can also control the column widths.


```

<table>
<colgroup>
<col width="30%" />
<col width="70%" />
</colgroup>
<thead>
<tr class="header">
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td markdown="span">First column **fields**</td>
<td markdown="span">Some descriptive text. This is a markdown link to [Google](http://google.com). Or see [some link][mydoc_tags].</td>
</tr>
<tr>
<td markdown="span">Second column **fields**</td>
<td markdown="span">Some more descriptive text.
</td>
</tr>
</tbody>
</table>

```

Result:

Field	Description
First column fields	Some descriptive text. This is a markdown link to Google . Or see some link (page 73) .
Second column fields	Some more descriptive text.

jQuery DataTables

You also have the option of using a [jQuery DataTable](#) , which gives you some additional capabilities. To use a jQuery DataTable in a page, include `datatable: true` in a page's frontmatter. This tells the default layout to load the necessary CSS and javascript bits and to include a `$(document).ready()` function that initializes the DataTables library.

You can change the options used to initialize the DataTables library by editing the call to `$('#table.display').DataTable()` in the default layout. The available options for Datatables are described in the [DataTable documentation](#), which is excellent.

You also must add a class of `display` to your tables. You can change the class, but then you'll need to change the trigger defined in the `$(document).ready()` function in the default layout from `table.display` to the class you prefer.

You can also add page-specific triggers (by copying the `<script></script>` block from the default layout into the page) and classes, which lets you use different options on different tables.

If you use an HTML table, adding `class="display"` to the `<table>` tag is sufficient.

Markdown, however, doesn't allow you to add classes to tables, so you'll need to use a trick: add `<div class="datatable-begin"></div>` before the table and `<div class="datatable-end"></div>` after the table. The default layout includes a jQuery snippet that automagically adds the `display` class to any table it finds between those two markers. So you can start with this (we've trimmed the descriptions for display):

```
<div class="datatable-begin"></div>
```

Food sample type	Description	Category	Sample
-----	-----	-----	-----
Apples	A small, somewhat round ...	Fruit	Fuji
Bananas	A long and curved, often-yellow ...	Fruit	Snow
Kiwis	A small, hairy-skinned sweet ...	Fruit	Golden
Oranges	A spherical, orange-colored sweet ...	Fruit	Navel

```
<div class="datatable-end"></div>
```

and get this:

Food	Description	Category	Sample type
Apples	A small, somewhat round and often red-colored, crispy fruit grown on trees.	Fruit	Fuji
Bananas	A long and curved, often-yellow, sweet and soft fruit that grows in bunches in tropical climates.	Fruit	Snow
Kiwis	A small, hairy-skinned sweet fruit with green-colored insides and seeds.	Fruit	Golden
Oranges	A spherical, orange-colored sweet fruit commonly grown in Florida and California.	Fruit	Navel

Notice a few features:

- You can keyword search the table. When you type a word, the table filters to match your word.
- You can sort the column order.
- You can page the results so that you show only a certain number of values on the first page and then require users to click next to see more entries.

Read more of the [DataTable documentation](#) to get a sense of the options you can configure. You should probably only use DataTables when you have long, massive tables full of information.

❗ Note: Try to keep the columns to 3 or 4 columns only. If you add 5+ columns, your table may create horizontal scrolling with the theme. Additionally, keep the column heading titles short.

Syntax highlighting

Summary: You can apply syntax highlighting to your code. This theme uses pygments and applies color coding based on the lexer you specify.

About syntax highlighting

For syntax highlighting, use fenced code blocks optionally followed by the language syntax you want:

```
```java
import java.util.Scanner;

public class ScannerAndKeyboard
{
 public static void main(String[] args)
 {
 Scanner s = new Scanner(System.in);
 System.out.print("Enter your name: ");
 String name = s.nextLine();
 System.out.println("Hello " + name + "!");
 }
}
```
```

This looks as follows:

```
import java.util.Scanner;

public class ScannerAndKeyboard
{
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        System.out.print( "Enter your name: " );
        String name = s.nextLine();
        System.out.println( "Hello " + name + "!" );
    }
}
```

Fenced code blocks require a blank line before and after.

If you're using an HTML file, you can also use the `highlight` command with Liquid markup.

```
{% highlight java %}
import java.util.Scanner;

public class ScannerAndKeyboard
{
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        System.out.print( "Enter your name: " );
        String name = s.nextLine();
        System.out.println( "Hello " + name + "!" );
    }
}
{% endhighlight %}
```

Result:

```
import java.util.Scanner;

public class ScannerAndKeyboard
{
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        System.out.print( "Enter your name: " );
        String name = s.nextLine();
        System.out.println( "Hello " + name + "!" );
    }
}
```

The theme has syntax highlighting specified in the configuration file as follows:

```
highlighter: rouge
```

The syntax highlighting is done via the `css/syntax.css` file.

Available lexers

The keywords you must add to specify the highlighting (in the previous example, `ruby`) are called “lexers.” You can search for “lexers.” Here are some common ones I use:

- js
- html
- yaml
- css
- json
- php
- java
- cpp
- dotnet
- xml
- http

Workflow maps

Summary: Version 6.0 of the Documentation theme for Jekyll reverts back to relative links so you can view the files offline. Additionally, you can store pages in subdirectories. Templates for alerts and images are available.

Workflow maps overview

You can implement workflow maps at the top of your pages. This is helpful if you're describing a process that involves multiple topics. See the following demos:

- [Simple workflow maps \(page 0\)](#)
- [Complex workflow maps \(page 0\)](#)

Simple workflow maps

1. Create an include at `_includes/custom/usermap.html`, where `usermap.html` contains the workflow and links you want. See the `usermap.html` as an example. It should look something like this:

```
<div id="userMap">
<div class="content"><a href="p2_sample1.html"><div clas
s="box box1">Connect to ADB</div></a></div>
<div class="arrow">→</div>
<div class="content"><a href="p2_sample2.html"><div clas
s="box box2">Download and Build the Starter Kit</di
v></a></div>
<div class="arrow">→</div>
<div class="content"><a href="p2_sample3.html"><div clas
s="box box3">Take a Tour</div></a></div>
<div class="arrow">→</div>
<div class="content"><a href="p2_sample4.html"><div clas
s="box box4">Load Your Widgets</div></a></div>
<div class="arrow">→</div>
<div class="content"><a href="p2_sample5.html"><div clas
s="box box5">Query for Something</div></a></div>
<div class="clearfix"></div>
</div>
```

You can have only 5 possible workflow squares across. Also, the links must be manually coded HTML like those shown, not automated Markdown links. (This is because the boxes are linked.)

2. Where you want the user maps to appear, add the sidebar properties shown in red below:

```
---
title: Sample 1 Topic
keywords: sample
summary: "This is just a sample topic..."
sidebar: product2_sidebar
permalink: p2_sample1
folder: product2
simple_map: true
map_name: usermap
box_number: 1
---
```

In the page.html layout, the following code gets activated when `simple_map` equals `true`:

```
{% if page.simple_map == true %}

<script>
  $(document).ready ( function(){
    $('.box{{page.box_number}}').addClass('active');
  });
</script>

{% include custom/{{page.map_name}}.html %}

{% endif %}
```

The script adds an `active` class to the box number, which automatically makes the active workflow box become highlighted based on the page you're viewing.

The `map_name` gets used as the name of the included file.

Complex workflow maps

The simpler user workflow allows for 5 workflow steps. If you have a more complex workflow, with multiple possible steps, branching, and more, consider using a complex workflow map. This map uses modals to show a list of instructions and links for each step.

1. Create an include at `_includes/custom/usermapcomplex.html`, where `usermapcomplex.html` contains the workflow and links you want. See the `usermapcomplex.html` as an example. The code in that file simply implements Bootstrap modals to create the pop-up boxes. Add your custom content inside the modal body:

```
<div class="modal-body">
<p>This is just dummy text ... Your first steps should be
to get started. You will need to do the following:</p>

  <ul>
    <li><a href="p2_sample6.html">Sample 6</a></li>
    <li><a href="p2_sample7.html">Sample 7</a></li>
    <li><a href="p2_sample8.html">Sample 8</a></li>
  </ul>
  <p>If you run into any of these setup issues, you must
  solve them before you can continue on.</p>

</div>
```

The existing `usermapcomplex.html` file just has 3 workflow square modals. If you need more, duplicate the modal code. In the duplicated code, make sure you make the following values in red unique (but the same within the same modal):

```
<button type="button" class="btn btn-default btn-lg modalButton3" data-toggle="modal" data-target="#myModal3">Publish your app</button>

  <div class="modal fade" id="myModal3" tabindex="-1" role="dialog" aria-labelledby="myModalLabel">
```

2. For each topic where you want the modal to appear, insert the following properties in your frontmatter:

```
---
title: Sample 6 Topic
keywords: sample
summary: "This is just a sample topic..."
sidebar: product2_sidebar
permalink: p2_sample6
complex_map: true
map_name: usermapcomplex
box_number: 1
toc: false
folder: product2
---
```

When your frontmatter contains `complex_map` equal to `true`, the following code gets activated in the `page.layout.html` file:

In the `page.html` layout, the following code gets activated when ``map`` equals ``true``:

```
{% if page.complex_map == true %}

<script>
  $(document).ready ( function(){
    $('.modalButton{{page.box_number}}').addClass('active');
  });
</script>

{% include custom/{{page.map_name}}.html %}

{% endif %}
```
```

# Commenting on files

**Summary:** You can add a button to your pages that allows people to add comments.

## About the review process

If you're using the doc as code approach, you might also consider using the same techniques for reviewing the doc as people use in reviewing code. This approach will involve using Github to edit the files.

There's an Edit me button on each page on this theme. This button allows collaborators to edit the content on Github.

Here's the code for that button on the page.html layout for GitHub:

```
{% if site.github_editme_path %}

<i class="fa fa-github fa-lg"></i> Edit me

{% endif %}
```

and here for GitLab:

```
{% if site.gitlab_editme_path %}

<i class="fa fa-gitlab fa-lg"></i> Edit me

{% endif %}
```

In your configuration file, edit the value for `github_editme_path` (or for Gitlab: `gitlab_editme_path`). For example, you might create a branch called “reviews” on your Github repo. Then you would add something like this in your configuration

file for the 'github\_editme\_path': tomjoht/documentation-theme-jekyll/edit/reviews. Here "tomjoht" is my github account name. The repo name is "documentation-theme-jekyll". The "reviews" name is the branch.

To suppress this button, comment out the `github_editme_path` in the `_config.yml` file.

## Add reviewers as collaborators

If you want people to collaborate on your project so that their edits get committed to a branch on your project, you need to add them as collaborators. For your Github repo, click **Settings** and add the collaborators on the Collaborators tab using their Github usernames.

If you don't want to allow anyone to commit to your Github branch, don't add the reviewers as collaborators. When someone makes an edit, Github will fork the theme. The person's edit then will appear as a pull request to your repo. You can then choose to merge the change indicated in the pull or not.

**Note:** When you process pull requests, you have to accept everything or nothing. You can't pick and choose which changes you'll merge. Therefore you'll probably want to edit the branch you're planning to merge or ask the contributor to make some changes to the fork before processing the pull request.

## Workflow

Users will make edits in your "reviews" branch (or whatever you want to call it). You can then commit those edits as you make updates.

When you're finished making all updates in the branch, you can merge the branch into the master.

Note that if you're making updates online, those updates will be out of sync with any local edits.

**Warning:** Don't make edits both online using Github's browser-based interface AND offline on your local machine using your local tools. When you try to push from your local, you'll likely get a merge conflict error. Instead, make sure you do a pull and update on your local after making any edits online.

## Prose.io

Prose.io is an overlay on Github that would allow people to make comments in an easier interface. If you simply go to [prose.io](https://prose.io), it asks to authorize your Github account, and so it will read files directly from Github but in the Prose.io interface.

## Build arguments

**Summary:** You use various build arguments with your Jekyll project. You can also create shell scripts to act as shortcuts for long build commands. You can store the commands in iTerm as profiles as well.

### How to build Jekyll sites

The normal way to build the Jekyll site is through the build command:

```
jeekyll build
```

To build the site and view it in a live server so that Jekyll rebuilds that site each time you make a change, use the `serve` command:

```
jeekyll serve
```

By default, the `_config.yml` in the root directory will be used, Jekyll will scan the current directory for files, and the folder `_site` will be used as the output. You can customize these build commands like this:

```
jeekyll serve --config configs/myspecialconfig.yml --destination ../doc_outputs
```

Here the `configs/myspecialconfig.yml` file is used instead of `_config.yml`. The destination directory is `../doc_outputs`, which would be one level up from your current directory.

### Shortcuts for the build arguments

If you have a long build argument and don't want to enter it every time in Jekyll, noting all your configuration details, you can create a shell script and then just run the script. Simply put the build argument into a text file and save it with the `.sh` extension (for Mac) or `.bat` extension (for Windows). Then run it like this:

```
. myscript.sh
```

My preference is to add the scripts to profiles in iTerm. See [iTerm Profiles \(page 157\)](#) for more details.

## Stop a server

When you're done with the preview server, press **Ctrl+C** to exit out of it. If you exit iTerm or Terminal without shutting down the server, the next time you build your site, or if you build multiple sites with the same port, you may get a server-already-in-use message.

You can kill the server process using these commands:

```
ps aux | grep jekyll
```

Find the PID (for example, it looks like “22298”).

Then type `kill -9 22298` where “22298” is the PID.

To kill all Jekyll instances, use this:

```
kill -9 $(ps aux | grep '[j]ekyll' | awk '{print $2}')
```

I recommend creating a profile in iTerm that stores this command. Here's what the iTerm settings look like:

The screenshot shows the iTerm2 preferences window with the 'General' tab selected. The settings are as follows:

- Basics**
  - Name: Kill all Jekyll
  - Shortcut key: ^⌘ (Command key)
  - Tags: Example: linux, dark bg, tall window
- Command**
  - ☒ Login shell
  - ☐ Command:
  - Send text at start: kill -9 \$(ps aux | grep '[j]ekyll' | awk '{print \$2}')
- Working Directory**
  - ☐ Home directory
  - ☐ Reuse previous session's directory
  - ☒ Directory: /Users/tjohnson/projects/docs
  - ☐ Advanced Configuration [Edit...](#)
- URL Schemes**
  - Schemes handled: Select URL Schemes...

*iTerm profile settings to kill all Jekyll*



# Themes

**Summary:** You can choose between two different themes (one green, the other blue) for your projects. The theme CSS is stored in the CSS folder and configured in the configuration file for each project.

**Note:** The [gem-based theme](#) approach is not yet integrated into this theme.

## Theme options

You can choose a green or blue theme, or you can create your own. In the css folder, there are two theme files: theme-blue.css and theme-green.css. These files have the most common CSS elements extracted in their own CSS file. Just change the hex colors to the ones you want.

In the \_includes/head.html file, specify the theme file you want the output to use — for example, `theme_file: theme-green.css`. See this line:

```
<link rel="stylesheet" href="css/theme-green.css" />
```

## Theme differences

The differences between the themes is fairly minimal. The main navigation bar, sidebar, buttons, and heading colors change color. That's about it.

In a more sophisticated theming approach, you could use Sass files to generate rules based on options set in a data file, but I kept things simple here.

# Generating PDFs

**Summary:** You can generate a PDF from your Jekyll project. You do this by creating a web version of your project that is printer friendly. You then use utility called Prince to iterate through the pages and create a PDF from them. It works quite well and gives you complete control to customize the PDF output through CSS, including page directives and dynamic tags from Prince.

## PDF overview

This process for creating a PDF relies on Prince XML to transform the HTML content into PDF. Prince costs about \$500 per license. That might seem like a lot, but if you're creating a PDF, you're probably working for a company that sells a product, so you likely have access to some resources. There's also a free license that prints a small "P" watermark on your title page, so if you're fine with that, great.

The basic approach is to generate a list of all web pages that need to be added to the PDF, and then add leverage Prince to package them up into a PDF. Once you set it up, building a pdf is just a matter of running a couple of commands. Also, creating a PDF this way gives you a lot more control and customization capabilities than with other methods for creating PDFs. If you know CSS, you can entirely customize the output.

## Demo

You can see an example of the finished product here:

 PDF Download

To generate the PDF, browse to the theme's directory in your terminal and run this script:

```
• pdf-mydoc.sh
```

This builds a PDF for the documentation in the theme. Look in the **pdf** folder for the output, and see the "last generated date" to confirm that you generated the PDF.

To build a PDF for the other sample projects, run these commands:

```
• pdf-product1.sh
```

or

```
• pdf-product2.sh
```

You can see the details of the script in these files in the theme's root directory. For example, open pdf-mydoc.sh. It contains the following:

```
Note that .sh scripts work only on Mac. If you're on Window
s, install Git Bash and use that as your client.

echo 'Kill all Jekyll instances'
kill -9 $(ps aux | grep '[j]ekyll' | awk '{print $2}')
clear

echo "Building PDF-friendly HTML site for Mydoc ...";
bundle exec jekyll serve --detach --config _config.yml,pdfconfi
gs/config_mydoc_pdf.yml;
echo "done";

echo "Building the PDF ...";
prince --javascript --input-list=_site/pdfconfigs/prince-list.t
xt -o pdf/mydoc.pdf;

echo "Done. Look in the pdf directory to see if it printed succ
essfully."
```

After stopping all Jekyll instances, we build Jekyll using a special configuration file that specifies a unique stylesheet. The build contains a file (prince-list.txt) that contains a list of all pages to be included in the PDF. We feed this list into a Prince command to build the PDF.

The following sections explain more about the setup.

## 1. Set up Prince

Download and install [Prince](#).

You can install a fully functional trial version. The only difference is that the title page will have a small Prince PDF watermark.

## 2. Create a new configuration file for each of your PDF targets

The PDF configuration file will build on the settings in the regular configuration file but will have some additional fields. Here's the configuration file for the mydoc product within this theme. This configuration file is located in the pdfconfigs folder.

```
destination: _site/
url: "http://127.0.0.1:4010"
baseurl: "/mydoc-pdf"
port: 4010
output: pdf
product: mydoc
print_title: Jekyll theme for documentation – mydoc product
print_subtitle: version 5.0
output: pdf
defaults:
 -
 scope:
 path: ""
 type: "pages"
 values:
 layout: "page_print"
 comments: true
 search: true

pdf_sidebar: mydoc_sidebar
```

**Note:** Although you're creating a PDF, you must still build an HTML web target before running Prince. Prince will pull from the HTML files and from the file-list for the TOC.

Note that the default page layout specified by this configuration file is `page_print`. This layout strips out all the sections that shouldn't appear in the print PDF, such as the sidebar and top navigation bar.

Also note that there's a `output: pdf` property in case you want to make some of your content unique to PDF output. For example, you could add conditional logic that checks whether `site.output` is `pdf` or `web`. If it's `pdf`, then include information only for the PDF, and so on. If you're using nav tabs, you'll definitely want to create an alternative experience in the PDF.

In the configuration file, customize the values for the `print_title` and `print_subtitle` that you want. These will appear on the title page of the PDF.

We will access this configure file in the PDF generation script.

### 3. Make sure your sidebar data file has titlepage.html and tocpage.html entries

There are two template pages in the root directory that are critical to the PDF:

- titlepage.html
- tocpage.html

These pages should appear in your sidebar YML file (in this product, `mydoc_sidebar.yml`):

```
- title:
 output: pdf
 type: frontmatter
 folderitems:
 - title:
 url: /titlepage.html
 output: pdf
 type: frontmatter
 - title:
 url: /tocpage.html
 output: pdf
 type: frontmatter
```

Leave these pages here in your sidebar. (The `output: pdf` property means they won't appear in your online TOC because the conditional logic of the sidebar.html checks whether `web` is equal to `pdf` or not before including the item in the web version of the content.)

The code in the tocpage.html is mostly identical to that of the sidebar.html page. This is essential for Prince to create the page numbers correctly with cross references.

There's another file (in the root directory of the theme) that is critical to the PDF generation process: `prince-list.txt`. This file simply iterates through the items in your sidebar and creates a list of links. Prince will consume the list of links from `prince-list.txt` and create a running PDF that contains all of the pages listed, with appropriate cross references and styling for them all.

**Note:** If you have any files that you do not want to appear in the PDF, add `output: web` (rather than `output: pdf`) in the list of attributes in your sidebar. The `prince-list.txt` file that loops through the `mydoc_sidebar.yml` file to grab the URLs of each page that should appear in the PDF will skip over any items that do not list `output: pdf` in the item attributes. For example, you might not want your tag archives to appear in the PDF, but you probably will want to list them in the online help navigation.

## 4. Customize your headers and footers

Open up the `css/printstyles.css` file and customize what you want for the headers and footers. At the very least, customize the email address ( `youremail@domain.com` ) that appears in the bottom left.

Exactly how the print styling works here is pretty nifty. You don't need to understand the rest of the content in this section unless you want to customize your PDFs to look different from what I've configured. But I'm adding this information here in case you want to understand how to customize the look and feel of the PDF output.

This style creates a page reference for a link:

```
a[href]::after {
 content: " (page " target-counter(attr(href), page) ")"
}
```

You don't want cross references for any link that doesn't reference another page, so this style specifies that the content after should be blank:

```
a[href*="mailto"]::after, a[data-toggle="tooltip"]::after, a[href].noCrossRef::after {
 content: "";
}
```

✓ **Tip:** If you have a link to a file download, or some other link that shouldn't have a cross reference (such as link used in JavaScript for navtabs or collapsible sections, for example, add `noCrossRef` as a class to the link to avoid having it say "page 0" in the cross reference.

This style specifies that after links to web resources, the URL should be inserted instead of the page number:

```
a[href^="http:"]::after, a[href^="https:"]::after {
 content: " (" attr(href) ")";
}
```

This style sets the page margins:

```
@page {
 margin: 60pt 90pt 60pt 90pt;
 font-family: sans-serif;
 font-style:none;
 color: gray;
}
```

To set a specific style property for a particular page, you have to name the page. This allows Prince to identify the page.

First you add frontmatter to the page that specifies the type. For the `titlepage.html`, here's the frontmatter:

```

type: title

```

For the `tocpage`, here's the frontmatter:

```

type: frontmatter

```

For the index.html page, we have this type tag (among others):

```

type: first_page

```

The default\_print.html layout will change the class of the `body` element based on the type value in the page's frontmatter:

```
<body class="{% if page.type == "title"%}title{% elsif page.type == "frontmatter" %}frontmatter{% elsif page.type == "first_page" %}first_page{% endif %} print">
```

Now in the `css/printstyles.css` file, you can assign a page name based on a specific class:

```
body.title { page: title }
```

This means that for content inside of `body class="title"`, we can style this page in our stylesheet using `@page title`.

Here's how that title page is styled:

```
@page title {
 @top-left {
 content: " ";
 }
 @top-right {
 content: " "
 }
 @bottom-right {
 content: " ";
 }
 @bottom-left {
 content: " ";
 }
}
```

As you can see, we don't have any header or footer content, because it's the title page.



For the `tocpage.html`, which has the `type: frontmatter`, this is specified in the stylesheet:

```
body.frontmatter { page: frontmatter }
body.frontmatter {counter-reset: page 1}

@page frontmatter {
 @top-left {
 content: prince-script(guideName);
 }
 @top-right {
 content: prince-script(datestamp);
 }
 @bottom-right {
 content: counter(page, lower-roman);
 }
 @bottom-left {
 content: "youremail@domain.com"; }
}
```

With `counter(page, lower-roman)`, we reset the page count to 1 so that the title page doesn't start the count. Then we also add some header and footer info. The page numbers start counting in lower-roman numerals.

Finally, for the first page (which doesn't have a specific name), we restart the counting to 1 again and this time use regular numbers.

```
body.first_page {counter-reset: page 1}

h1 { string-set: doctitle content() }

@page {
 @top-left {
 content: string(doctitle);
 font-size: 11px;
 font-style: italic;
 }
 @top-right {
 content: prince-script(datestamp);
 font-size: 11px;
 }

 @bottom-right {
 content: "Page " counter(page);
 font-size: 11px;
 }
 @bottom-left {
 content: prince-script(guideName);
 font-size: 11px;
 }
}
```

You'll see some other items in there such as `prince-script`. This means we're using JavaScript to run some functions to dynamically generate that content. These JavaScript functions are located in the `_includes/head_print.html`:

```
<script>
 Prince.addScriptFunc("datestamp", function() {
 return "PDF last generated: October 28, 2021";
 });
</script>

<script>
 Prince.addScriptFunc("guideName", function() {
 return "Jekyll theme for documentation – mydoc product
User Guide";
 });
</script>
```

There are a couple of Prince functions that are default functions from Prince. This gets the heading title of the page:

```
content: string(doctype);
```

This gets the current page:

```
content: "Page " counter(page);
```

Because the theme uses JavaScript in the CSS, you have to add the `--javascript` tag in the Prince command (detailed later on this page).

## 5. Customize and run the PDF script

Duplicate the `pdf-mydoc.sh` file in the root directory and customize it for your specific configuration files.

```
echo 'Killing all Jekyll instances'
kill -9 $(ps aux | grep '[j]ekyll' | awk '{print $2}')
clear

echo "Building PDF-friendly HTML site for Mydoc ...";
jekyll serve --detach --config _config.yml,pdfconfigs/config_mydoc_pdf.yml;
echo "done";

echo "Building the PDF ...";
prince --javascript --input-list=_site/pdfconfigs/prince-list.txt -o pdf/mydoc.pdf;
echo "done";
```

Note that the first part kills all Jekyll instances. This way you won't try to serve Jekyll at a port that is already occupied.

The `jekyll serve` command serves up the HTML-friendly PDF configurations for our two projects. This web version is where Prince will go to get its content.

The prince script issues a command to the Prince utility. JavaScript is enabled (`--javascript`), and we tell it exactly where to find the list of files (`--input-list`) — just point to the `prince-list.txt` file. Then we tell it where and what to output (`-o`).

Make sure that the path to the `prince-list.txt` is correct. For the output directory, I like to output the PDF file into my project's source (into the `files` folder). Then when I build the web output, the PDF is included and something I can refer to.

**Note:** You might not want to include the PDF in your project files, since you're likely committing the PDF to Github and as a result saving the changes from one PDF version to another with each save.

## 6. Add conditions for your new builds in the PDF config file

In the PDF configuration file, there's a section that looks like this:

```
{% if site.product == "mydoc" %}
pdf_sidebar: product2_sidebar
{% endif %}
```

Point to the sidebar you want here.

What this does is allow the prince-list.txt and toc.html files to iterate through the right sidebar. Otherwise, you would need to create a unique prince-list.txt and toc.html file for each separate PDF output you have.

## 7. Add a download button for the PDF

You can add a download button for your PDF using some Bootstrap button code:

```
<button type="button" class="btn btn-default" aria-label="Left Align"> PDF Download</button>
```

Here's what that looks like:

```

```

 PDF Download </a>

## JavaScript conflicts

If you have JavaScript on any of your pages, Prince will note errors in Terminal like this:

```
error: TypeError: value is not an object
```

However, the PDF will still build.

You need to conditionalize out any JavaScript from your PDF web output before building your PDFs. Make sure that the PDF configuration files have the `output: pdf` property.

Then surround the JavaScript with conditional tags like this:

```
{% raw %}{% unless site.output == "pdf" %}
javascript content here ...
{% endunless %}
```

For more detail about using `unless` in conditional logic, see [Conditional logic \(page 45\)](#). What this code means is “run this code unless this value is the case.”

## Overriding Bootstrap Print Styles

The theme relies on Bootstrap’s CSS for styling. However, for print media, Bootstrap applies the following style:

```
@media print{*,:after,:before{color:#000!important;text-shado
w:none!important;background:0 0!important;-webkit-box-shadow:no
ne!important;box-shadow:none!important}}
```

This is minified, but basically the `*` (asterisk) means select all, and applied the color `#000` (black). As a result, the Bootstrap style strips out all color from the PDF (for Bootstrap elements).

This is problematic for code snippets that have syntax highlighting. I decided to remove this de-coloring from the print output. I commented out the Bootstrap style:

```
@media print{*,:after,:before{/*color:#000!important;*/text-sha
dow:none!important;*/background:0 0!important;*/-webkit-box-sha
dow:none!important;box-shadow:none!important}}
```

If you update Bootstrap, make sure you make this edit. (Sorry, admittedly I couldn’t figure out how to simply overwrite the `*` selector with a later style.)

I did, however, remove the color from the alerts and lighten the background shading for `pre` elements. The `printstyles.css` has this setting.

# Help APIs and UI tooltips

**Summary:** You can loop through files and generate a JSON file that developers can consume like a help API. Developers can pull in values from the JSON into interface elements, styling them as popovers for user interface text, for example. The beauty of this method is that the UI text remains in the help system (or at least in a single JSON file delivered to the dev team) and isn't hard-coded into the UI.

## Full code demo of content API

You can create a help API that developers can use to pull in content.

For the full code demo, see the notes in the [Tooltips file \(page 0\)](#).

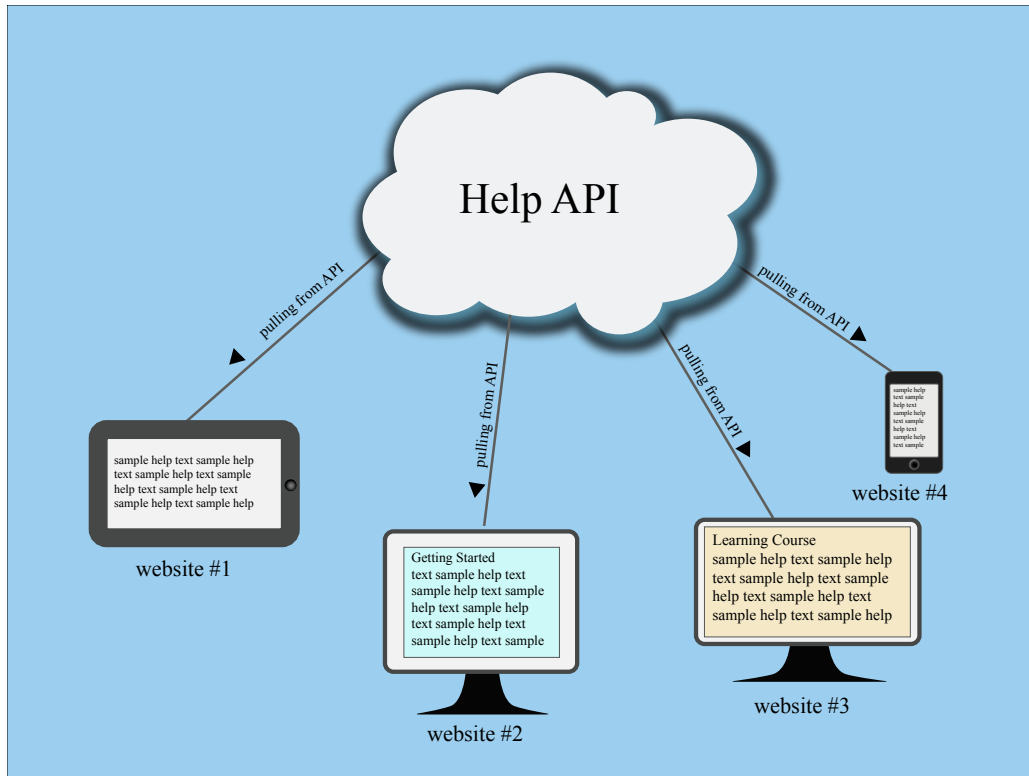
In this demo, the popovers pull in and display content from the information in a [tooltips.json \(page 0\)](#) file located in the same directory.

Instead of placing the JSON source in the same directory, you could also host the JSON file on another site.

Additionally, instead of tooltip popovers, you could also print content directly to the page. Basically, whatever you can stuff into a JSON file, developers can integrate it onto a page.

## Diagram overview

Here's a diagram showing the basic idea of the help API:



Is this really an API? Well, sort of. The help content is pushed out into a JSON file that other websites and applications can easily consume. The endpoints don't deliver different data based on parameters added to a URL. But the overall concept is similar to an API: you have a client requesting resources from a server.

Note that in this scenario, the help is openly accessible on the web. If you have a private system, it's more complicated.

To deliver help this way using Jekyll, follow the steps in each of the sections below.

## 1. Create a “collection” for the help content

A collection is another content type that extends Jekyll beyond the use of pages and posts. Call the collection “tooltips.”

Add the following information to your configuration file to declare your collection:

```
collections:
 tooltips:
 output: false
```



In your Jekyll project's root directory, create a new folder called “\_tooltips” and put every page that you want to be part of that tooltips collection inside that folder.

In Jekyll, folders that begin with an underscore (“\_”) aren't included in the output. However, in the collection information that you add to your configuration file, if you change `output` to `true`, the tooltips folder will appear in the output, and each page inside tooltips will be generated. You most likely don't want this for tooltips (you just want the JSON file), so make the `output` setting `false`.

## 2. Create tooltip definitions in a YAML file

Inside the `_data` folder, create a YAML file called something like `definitions.yml`. Add the definitions for each of your tooltips here like this:

```
basketball: "Basketball is a sport involving two teams of five
players each competing to put a ball through a small circular r
im 10 feet above the ground. Basketball requires players to be
in top physical condition, since they spend most of the game ru
nning back and forth along a 94-foot-long floor."
```

The definition of basketball is stored this data file so that you can re-use it in other parts of the help as well. You'll likely want the definition to appear not only in the tooltip in the UI, but also in the regular documentation as well.

## 3. Create pages in your collection

Create pages inside your new tooltips collection (that is, inside the `_tooltips` folder). Each page needs to have a unique `id` in the frontmatter as well as a `product`. Then reference the definition you created in the `definitions.yml` file.

Here's an example:

```

doc_id: basketball
product: mydoc

{{site.data.definitions.basketball}}
```

(Note: Avoid using `id`, as it seems to generate out as `/tooltips/basketball` instead of just ``basketball``.)

You need to create a separate file for each tooltip you want to deliver.

The product attribute is required in the frontmatter to distinguish the tooltips produced here from the tooltips for other products in the same `_tooltips` folder. When creating the JSON file, Jekyll will iterate through all the pages inside `_tooltips`, regardless of any subfolders included here.

## 4. Create a JSON file that loops through your collection pages

Now it's time to create a JSON file with Liquid code that iterates through our tooltip collection and grabs the information from each tooltip file.

Inside your project's pages directory (e.g., `mydoc`), add a file called `"tooltips.json"`. (You can use whatever name you want.) Add the following to your JSON file:

```

layout: null
search: exclude

{
 "entries":
 [
 {% for page in site.tooltips %}
 {
 "doc_id": "{{ page.doc_id }}",
 "body": "{{ page.content | strip_newlines | replace: '\',
 '\\\\' | replace: '\"', '\\\"' }}"
 } {% unless forloop.last %},{% endunless %}
 {% endfor %}
]
}
```

This code will loop through all pages in the `tooltips` collection and insert the `id` and `body` into key-value pairs for the JSON code. Here's an example of what that looks like after it's processed by Jekyll in the site build:

```
{
 "entries": [
 {
 "doc_id": "baseball",
 "body": "Baseball is considered America's pasttime sport, though that may be more of a historical term than a current one. There's a lot more excitement about football than baseball. A baseball game is somewhat of a snooze to watch, for the most part."
 },
 {
 "doc_id": "basketball",
 "body": "Basketball is a sport involving two teams of five players each competing to put a ball through a small circular rim 10 feet above the ground. Basketball requires players to be in top physical condition, since they spend most of the game running back and forth along a 94-foot-long floor."
 },
 {
 "doc_id": "football",
 "body": "No doubt the most fun sport to watch, football also manages to accrue the most injuries with the players. From concussions to blown knees, football players have short sport lives."
 },
 {
 "doc_id": "soccer",
 "body": "If there's one sport that dominates the world landscape, it's soccer. However, US soccer fans are few and far between. Apart from the popularity of soccer during the World Cup, most people don't even know the name of the professional soccer organization in their area."
 }
]
}
```

You can also view the same JSON file here: [tooltips.json \(page 0\)](#).

You can add different fields depending on how you want the JSON to be structured. Here we just have two fields: `doc_id` and `body`. And the JSON is looking just in the tooltips collection that we created.

✓ **Tip:** Check out [Google's style guide](#) for JSON. These best practices can help you keep your JSON file valid.

Note that you can create different JSON files that specialize in different content. For example, suppose you have some getting started information. You could put that into a different JSON file. Using the same structure, you might add an `if` tag that checks whether the page has frontmatter that says `type: getting_started` or something. Or you could put the content into separate collection entirely (different from tooltips).

By chunking up your JSON files, you can provide a quicker lookup. (I'm not sure how big the JSON file can be before you experience any latency with the jQuery lookup.)

## 5. Build your site and look for the JSON file

When you build your site, Jekyll will iterate through every page in your `_tooltips` folder and put the page id and body into this format. In the output, look for the JSON file in the `tooltips.json` file. You'll see that Jekyll has populated it with content. This is because of the triple hyphen lines in the JSON file — this instructs Jekyll to process the file.

## 6. Allow CORS access to your help if stored on a remote server

You can simply deliver the JSON file to devs to add to the project. But if you have the option, it's best to keep the JSON file stored in your own help system. Assuming you have the ability to update your content on the fly, this will give you completely control over the tooltips without being tied to a specific release window.

When people make calls to your site *from other domains*, you must allow them access to get the content. To do this, you have to enable something called CORS (cross origin resource sharing) within the server where your help resides.

In other words, people are going to be executing calls to reach into your site and grab your content. Just like the door on your house, you have to unlock it so people can get in. Enabling CORS is unlocking it.

How you enable CORS depends on the type of server.

If your server setup allows `htaccess` files to override general server permissions, create an `.htaccess` file and add the following:

```
Header set Access-Control-Allow-Origin "*"
```

Store this in the same directory as your project. This is what I've done in a directory on my web host (bluehost.com). Inside `http://idratherassets.com/wp-content/apidemos/`, I uploaded a file called ".htaccess" with the preceding code.

After I uploaded it, I renamed it to .htaccess, right-clicked the file and set the permissions to 774.

To test whether your server permissions are set correctly, open a terminal and run the following curl command pointing to your tooltips.json file:

```
curl -I http://idratherassets.com/wp-content/apidemos/tooltips.json
```

The `-I` command tells cURL to return the request header only.

If the server permissions are set correctly, you should see the following line somewhere in the response:

```
Access-Control-Allow-Origin: *
```

If you don't see this response, CORS isn't allowed for the file.

If you have an AWS S3 bucket, you can supposedly add a CORS configuration to the bucket permissions. Log into AWS S3 and click your bucket. On the right, in the Permissions section, click **Add CORS Configuration**. In that space, add the following policy:

```
<CORSConfiguration>
 <CORSRule>
 <AllowedOrigin>*</AllowedOrigin>
 <AllowedMethod>GET</AllowedMethod>
 </CORSRule>
</CORSConfiguration>
```

(Although this should work, in my experiment it doesn't. And I'm not sure why...)

In other server setups, you may need to edit one of your Apache configuration files. See [Enable CORS](#) or search online for ways to allow CORS for your server.

If you don't have CORS enabled, users will see a CORS error/warning message in the console of the page making the request.

✓ **Tip:** If enabling CORS is problematic, you could always just send developers the tooltips.json file and ask them to place it on their own server.

## 7. Explain how developers can access the help

Developers can access the help using the `.get` method from jQuery, among other methods. Here's an example of how to get tooltips for basketball, baseball, football, and soccer:

```
var url = "tooltips.json";

$.get(url, function(data) {

 /* Bootstrap popover text is defined inside a data-content attribute inside an element. That's why I'm using attr here. If you just want to insert content on the page, use append and remove the data-content argument from the parentheses.*/

 $.each(data.entries, function(i, page) {
 if (page.doc_id == "basketball") {
 $("#basketball").attr("data-content", page.body);
 }

 if (page.doc_id == "baseball") {
 $("#baseball").attr("data-content", page.body);
 }

 if (page.doc_id == "football") {
 $("#football").attr("data-content", page.body);
 }

 if (page.doc_id == "soccer") {
 $("#soccer").attr("data-content", page.body);
 }

 });
});
```

View the [tooltip demo](#) for a demonstration. See the source code for full code details.

The `url` in the demo is relative, but you could equally point it to an absolute path on a remote host assuming CORS is enabled on the host.

The `each` method looks through all the JSON content to find the item whose `page.id` is equal to `basketball`. It then looks for an element on the page named `#basketball` and adds a `data-content` attribute to that element.

**⚠ Warning:** Make sure your JSON file is valid. Otherwise, this method won't work. I use the [JSON Formatter extension for Chrome](#). When I go to the `tooltips.json` page in my browser, the JSON content — if valid — is nicely formatted (and includes some color coding). If the file isn't valid, it's not formatted and there isn't any color. You can also check the JSON formatting using [JSON Formatter and Validator](#). If your JSON file isn't valid, identify the problem area using the validator and troubleshoot the file causing issues. It's usually due to some code that isn't escaping correctly.

Why `data-content`? Well, in this case, I'm using [Bootstrap popovers](#) to display the tooltip content. The `data-content` attribute is how Bootstrap injects popovers.

Here's the section on the page where the popover is inserted:

```
<p>Basketball </p>
```

Notice that I just have `id="basketball"` added to this popover element. Developers merely need to add a unique ID to each tooltip they want to pull in the help content. Either you tell developers the unique ID they should add, or ask them what IDs they added (or just tell them to use an ID that matches the field's name).

In order to use jQuery and Bootstrap, you'll need to add the appropriate references in the head tags of your page:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css">
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/js/bootstrap.min.js"></script>

<script type="text/javascript">
$(document).ready(function(){
 $('[data-toggle="popover"]').popover({
 placement : 'right',
 trigger: 'hover',
 html: true
 });
</script>
```

Again, see the [Tooltip Demo](#) for a demo of the full code.

Note that even though you reference a Bootstrap JS script, Bootstrap's popovers require you to initialize them using the above code as well — they aren't turned on by default.

View the source code of the [tooltip demo](#) for the full comments.

## 8. Create easy links to embed the help in your help site

You might also want to insert the same content into different parts of your help site. For example, if you have tooltips providing definitions for fields, you'll probably want to create a page in your help that lists those same definitions.

You could use the same method developers use to pull help content into their applications. But it will probably be easier to simply use Jekyll's tags for doing it.

Here's how you would reuse the content:



```
<h2>Reuse Demo</h2>

<table>
<thead>
<tr>
<th>Sport</th>
<th>Comments</th>
</tr>
</thead>
<tbody>

<tr>
<td>Basketball</td>
<td>{{site.data.definitions.basketball}}</td>
</tr>

<tr>
<td>Baseball</td>
<td>{{site.data.definitions.baseball}}</td>
</tr>

<tr>
<td>Football</td>
<td>{{site.data.definitions.football}}</td>
</tr>

<tr>
<td>Soccer</td>
<td>{{site.data.definitions.soccer}}</td>
</tr>
</tbody>
</table>
```

And here's the code:

## Reuse Demo

Sport	Comments
Basketball	Basketball is a sport involving two teams of five players each competing to put a ball through a small circular rim 10 feet above the ground. Basketball requires players to be in top physical condition, since they spend most of the game running back and forth along a 94-foot-long floor.
Baseball	Baseball is considered America's pasttime sport, though that may be more of a historical term than a current one. There's a lot more excitement about football than baseball. A baseball game is somewhat of a snooze to watch, for the most part.
Football	No doubt the most fun sport to watch, football also manages to accrue the most injuries with the players. From concussions to blown knees, football players have short sport lives.
Soccer	If there's one sport that dominates the world landscape, it's soccer. However, US soccer fans are few and far between. Apart from the popularity of soccer during the World Cup, most people don't even know the name of the professional soccer organization in their area.

Now you have both documentation and UI tooltips generated from the same definitions file.

# Search configuration

**Summary:** The search feature uses JavaScript to look for keyword matches in a JSON file. The results show instant matches, but it doesn't provide a search results page like Google. Also, sometimes invalid formatting can break the JSON file.

## About search

The search is configured through the search.json file in the root directory. The search is a simple search that looks at content in pages. It looks at titles, summaries, keywords, and tags.

However, the search doesn't work like google — you can't hit return and see a list of results on the search results page, with the keywords in bold. Instead, this search shows a list of page titles that contain keyword matches. It's fast, but simple.

## Excluding pages from search

By default, every page is included in the search. Depending on the type of content you're including, you may find that some pages will break the JSON formatting. If that happens, then the search will no longer work.

If you want to exclude a page from search add `search: exclude` in the page's frontmatter.

## Troubleshooting search

You should exclude any files from search that you don't want appearing in the search results. For example, if you have a tooltips.json file or prince-list.txt, don't include it, as the formatting will break the JSON format.

If any formatting in the search.json file is invalid (in the build), search won't work. You'll know that search isn't working if no results appear when you start typing in the search box.

If this happens, point your browser to your build's search.json file at <http://localhost:4000/search.json> (or your public github pages site), copy the entire search.json page as it's output on the browser, and then use a [JSON validator](#) to validate the output by pasting what you copied into the validator. Look for the line causing trouble. It will be highlighted. Edit the file that's causing

the problem to either exclude it from the search or fix the syntax so that it doesn't invalidate the JSON. (Note that tabs in the body will invalidate JSON, as will certain characters in the file's front matter. For example, the summary: cannot have " quotes in it.

The search.json file already tries to strip out content that would otherwise make the JSON invalid.

## Including the body field in search

I've found that including the `body` field in the search creates too many problems, and so I've removed `body` from the search. You can see the results of including the `body` by adding this along with the other fields in search.json:

```
"body": "{ { page.content | strip_html | strip_newlines |
replace: '\\', '\\\\' | replace: '\"', '\\\"' | replace: '
, ' ' } }",
```

Note that the last replace, `| replace: '^t', ' '`, looks for any tab character and replaces it with four spaces. (Tab characters invalidate JSON.) If you run into other problematic formatting, you can use regex expressions to find and replace the content. See [Regular Expressions](#) for details on finding and replacing code.

It's possible that the formatting may not account for all the scenarios that would invalidate the JSON. (Sometimes it's an extra comma after the last item that makes it invalid.)

Note that including the body in the search creates other problems as well. The search results show the most immediate matches in the JSON file. If several topics have matches for the keyword in the body, these matches might appear before other files that have matches in the title, summary, or keywords. This is because this simple search does not provide any weighting mechanisms for the content.

## Customizing search results

At some point, you may want to customize the search results more. Here's a little more detail that will be helpful. The search.json file retrieves various page values:

```

title: search
layout: none
search: exclude

[
{% for page in site.pages %}
{% unless page.search == "exclude" %}
{
"title": "{{ page.title | escape }}",
"tags": "{{ page.tags }}",
"keywords": "{{page.keywords}}",
"url": "{{ page.url | remove: "/" }}",
"summary": "{{page.summary | strip }}"
},
{% endunless %}
{% endfor %}

{% for post in site.posts %}

{
"title": "{{ post.title | escape }}",
"tags": "{{ post.tags }}",
"keywords": "{{post.keywords}}",
"url": "{{ post.url }}",
"summary": "{{post.summary | strip }}"
}
{% unless forloop.last %},{% endunless %}
{% endfor %}

]
```

The `_includes/topnav.html` file then makes use of these values:

```

 <!--start search-->
 <div id="search-demo-container">
 <input type="text" id="search-input" placeholder="search...">
 <ul id="results-container">
 </div>
 <script src="js/jekyll-search.js" type="text/javascript"></script>
 <script type="text/javascript">
 SimpleJekyllSearch.init({
 searchInput: document.getElementById('search-input'),
 resultsContainer: document.getElementById('results-container'),
 dataSource: 'search.json',
 searchResultTemplate: '{title}',
 noResultsText: 'No results found.',
 limit: 10,
 fuzzy: true,
 })
 </script>
 <!--end search-->

```

Where you see `{url}` and `{title}`, the search is retrieving the values for these as specified in the `search.json` file.

## More robust search

Overall, the built-in search only works for small documentation projects. If you have more robust search needs, consider integrating [Google Custom Search](#), [Algolia](#), or [Swifttype](#).

# iTerm profiles

**Summary:** You can set up profiles in iTerm to facilitate the build process with just a few clicks. This can make it a lot easier to quickly build multiple outputs.

## About iTerm profiles

When you're working with tech docs, a lot of times you have builds that push files onto different servers, or that build the content for different environments. It can be a hassle to type out these commands each time. Instead, it's easier to configure iTerm with profiles that initiate the scripts.

## Set up profiles

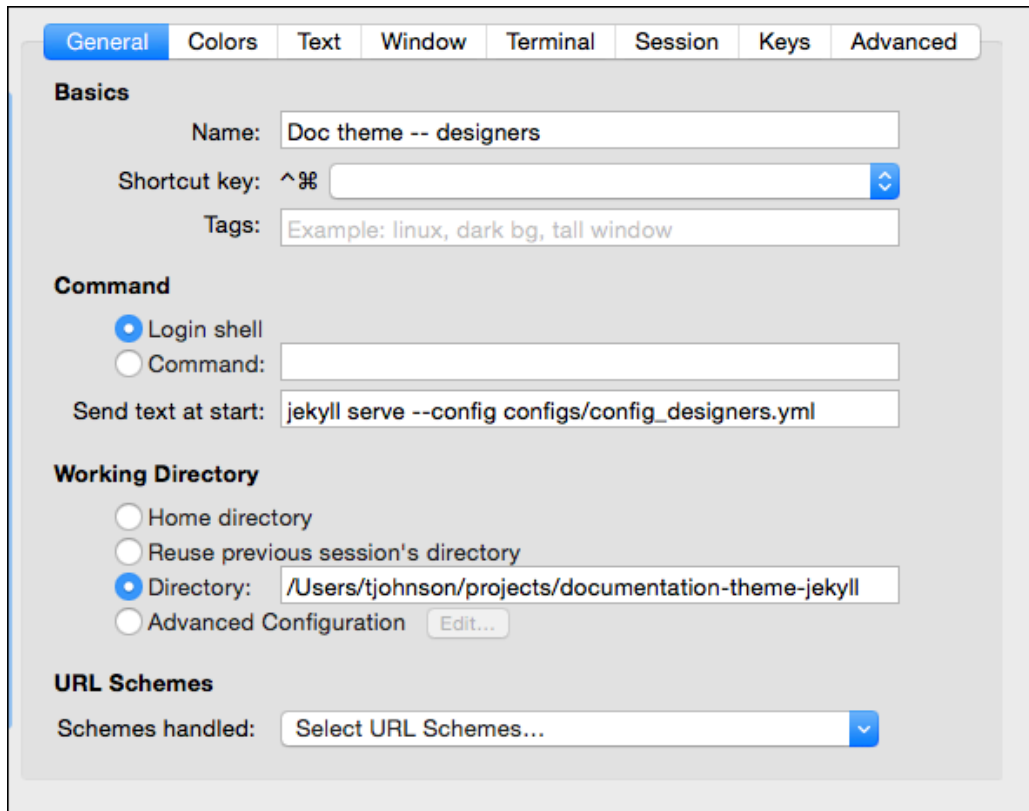
1. Open iTerm and go to **Profiles > Open Profiles**.
2. Click **Edit Profiles**.
3. Click the + button in the lower-left corner to create a new profile.
4. In the **Name** field, type a name describing the output, such as `Doc theme -- designers`.
5. In the **Send text at start** field, type the command for the build script, such as this:

```
JEKYLL_ENV=production jekyll serve
```

Leave the Login shell option selected.

6. In the Working Directory section, select **Directory** and enter the directory for your project, such as `/Users/tjohnson/projects/documentation-theme-jekyll`.
7. Close the profiles panel.

Here's an example:



The screenshot shows the 'General' tab of the iTerm profile configuration window. The tabs at the top are: General (selected), Colors, Text, Window, Terminal, Session, Keys, and Advanced. The 'Basics' section contains: Name: 'Doc theme -- designers', Shortcut key: '^⌘' (with a dropdown arrow), and Tags: 'Example: linux, dark bg, tall window'. The 'Command' section has radio buttons for 'Login shell' (selected) and 'Command:', with a text field below it containing 'jekyll serve --config configs/config\_designers.yml'. The 'Send text at start:' field is empty. The 'Working Directory' section has radio buttons for 'Home directory', 'Reuse previous session's directory', 'Directory:' (selected), and 'Advanced Configuration' (with an 'Edit...' button). The 'Directory:' field contains '/Users/tjohnson/projects/documentation-theme-jekyll'. The 'URL Schemes' section has a 'Schemes handled:' dropdown menu set to 'Select URL Schemes...'.

*iTerm profile example*

## Launching a profile

1. In iTerm, make sure the Toolbar is shown. Go to **View > Toggle Toolbar**.
2. Click the **New** button and select your profile.

✓ **Tip:** When you're done with the session, make sure to click **Ctrl+C**.



# Pushing builds to server

**Summary:** You can push your build to AWS using commands from the command line. By including your copy commands in commands, you can package all of the build and deploy process into executable scripts.

## Pushing to AWS S3

If you have the AWS Command Line Interface installed and are pushing your builds to AWS, the following commands show how you can build and push to an AWS location from the command line:

```
aws s3 cp ~/users/tjohnson/projects/mydocproject/ s3://[aws path]docpath/mydocproject --recursive

aws s3 cp ~/users/tjohnson/projects/anotherdocproject2/ s3://[aws path]docpath/anotherdocproject --recursive
```

The first path in the argument is the local location; the second path is the destination.

## Pushing to a regular server

If you're pushing to a regular server that you can ssh into, you can use `scp` commands to push your build. Here's an example:

```
scp -r /users/tjohnson/projects/mydocproject/ name@domain:/var/www/html/mydocproject
```

Similar to the above, the first path is the local location; the second path is the destination.

# Publishing on Github Pages

**Summary:** You can publish your project on Github Pages, which is a free web hosting service provided by Github. All you need is to put your content into a Github repo branch called gh-pages and make this your default branch in your repo. With a Jekyll site, you just commit your entire project into the gh-pages branch and Github Pages will build the site for you.

## Set up your Github repo

1. Make sure you have Git installed. You can download and install [Git for Windows here](#) and [Git for Mac here](#). If you're on a Mac, chances are you might already have git installed. You can check by opening up a terminal and typing `which git`.
2. Go to [Github.com](#) and sign up for an account.
3. Click the + button in the upper-right corner and select **New repository**.
4. Name the repository something like **mydocthem**.
5. Type a description..
6. Select the **Initialize this repository with a README** check box.
7. Add a license if desired.
8. Leave the other options at the defaults and click **Create repository**.
9. Click the **Settings** button.
10. Go to your repository's home page, and click the branch drop-down menu.
11. Create a new branch called **gh-pages**.
12. Click **Settings** and change the default branch to **gh-pages**.
13. Go back to your repository's homepage. With the gh-pages branch selected, copy the **https clone url**:
14. Open a terminal, browse to a convenient location for your project, and type `git clone https://github.com/tomjoht/myreponame.git`, replacing the `https://github.com/tomjoht/myreponame.git` with your repository's https clone URL that you copied.

15. Move the jekyll theme files into this new folder that you just created in the previous step.
16. Open the `_config.yml` file and add the following:

```
url: tomjoht.github.io
baseurl: /myreponame
```

Change the url to your github account name, and the baseurl to your repo name.

## Install Bundler

Bundler is a package manager for Ruby that will install all dependencies you might need to build your site locally. I recommend installing Bundler through homebrew. (Sorry, these instructions apply to Mac only.)

1. Install [homebrew](#):

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

2. Install Bundler:

```
gem install bundler
```

## Add the github pages gem

1. In terminal, browse to your Jekyll project directory.
2. Type `bundle init`. This creates a Gemfile and Gemfile.lock in your project.
3. Type `open gemfile`. This opens the gemfile in your default text editor.
4. Add the following in the gemfile (replacing the existing contents):

```
source 'https://rubygems.org'
gem 'github-pages'
```

5. Run `bundle install`.
6. Add the new jekyll files to git: `git add --all`.
7. Commit the files: `git commit -m "committing my jekyll theme"`.
8. Push the files up to your github repo: `git push`.

Github Pages will now automatically build your site. Wait a minute or two, and then visit `tomjoht.github.io/yourreponame`, replacing this path with your github account and branch.

## Customize your URL

You can also customize your Github URL. More instructions on this later....

## Knowledge-base layout

**Summary:** This shows a sample layout for a knowledge base. Each square could link to a tag archive page. In this example, font icons from Font Awesome are used for the graphics, and the layout is pulled from the Modern Business theme. .

Here's the sample knowledge-base style layout:

### Knowledge Base Categories



*Getting started*

Lorem ipsum dolor sit amet, consectetur adipisicing elit.

[Learn More \(page 0\)](#)



### Navigation

Lorem ipsum dolor sit amet, consectetur adipisicing elit.

[Learn More \(page 0\)](#)

### Single sourcing

Lorem ipsum dolor sit amet, consectetur adipisicing elit.

[Learn More \(page 0\)](#)

### Formatting

Lorem ipsum dolor sit amet, consectetur adipisicing elit.

[Learn More \(page 0\)](#)

## Generating a list of all pages with a certain tag

If you don't want to link to a tag archive index, but instead want to list all pages that have a certain tag, you could use this code:

Getting started pages:

```

{% assign sorted_pages = site.pages | sort: 'title' %}
{% for page in sorted_pages %}
{% for tag in page.tags %}
{% if tag == "getting_started" %}
{{page.titl
e}}
{% endif %}
{% endfor %}
{% endfor %}

```

Here's the result:

Getting started pages:

- [About Ruby, Gems, Bundler, and other prerequisites \(page 16\)](#)
- [About the theme's author \(page 10\)](#)
- [Blain's World @ ACC \(page 3\)](#)
- [Install Jekyll on Mac \(page 24\)](#)
- [Pages \(page 32\)](#)
- [Posts \(page 39\)](#)
- [Release notes 5.0 \(page 14\)](#)
- [Release notes 6.0 \(page 12\)](#)
- [Sidebar Navigation \(page 59\)](#)
- [Support \(page 11\)](#)
- [Supported features \(page 5\)](#)

# Glossary layout

**Summary:** Your glossary page can take advantage of definitions stored in a data file. This gives you the ability to reuse the same definition in multiple places. Additionally, you can use Bootstrap classes to arrange your definition list horizontally.

You can create a glossary for your content. First create your glossary items in a data file such as `glossary.yml`.

Then create a page and use definition list formatting, like this:

**fractious**

Like a little mischevious child, full of annoying and constant trouble.

**gratuitous**

Something that is unwarranted and uncouth, like the social equivalent of a flagrant foul.

**haughty**

Proud and flaunting it. Holding your head high up like a snooty, too-good-for-everything rich person.

**gratuitous**

Something that is unwarranted and uncouth, like the social equivalent of a flagrant foul.

**impertinent**

Brave and courageous especially in a difficult, dangerous situation.

Here's the code:



```
fractious
: {{site.data.glossary.fractious}}

gratuitous
: {{site.data.glossary.gratuitous}}

haughty
: {{site.data.glossary.haughty}}

gratuitous
: {{site.data.glossary.gratuitous}}

impertinent
: {{site.data.glossary.intrepid}}
```

The glossary works well as a link in the top navigation bar.

## Horizontally styled definiton lists

You can also change the definition list ( `dl` ) class to `dl-horizontal` . This is a Bootstrap specific class. If you do, the styling looks like this:

### **fractious**

Like a little mischevious child, full of annoying and constant trouble.

### **gratuitous**

Something that is unwarranted and uncouth, like the social equivalent of a flagrant foul.

### **haughty**

Proud and flaunting it. Holding your head high up like a snooty, too-good-for-everything rich person.

### **gratuitous**

Something that is unwarranted and uncouth, like the social equivalent of a

flagrant foul.

## impertinent

Someone acting rude and insensitive to others.

## intrepid

Brave and courageous especially in a difficult, dangerous situation.

For this type of list, you must use HTML. The list would then look like this:

```
<dl class="dl-horizontal">

 <dt id="fractious">fractious</dt>
 <dd>{{site.data.glossary.fractious}}</dd>

 <dt id="gratuitous">gratuitous</dt>
 <dd>{{site.data.glossary.gratuitous}}</dd>

 <dt id="haughty">haughty</dt>
 <dd>{{site.data.glossary.haughty}}</dd>

 <dt id="benchmark_id">gratuitous</dt>
 <dd>{{site.data.glossary.gratuitous}}</dd>

 <dt id="impertinent">impertinent</dt>
 <dd>{{site.data.glossary.impertinent}}</dd>

 <dt id="intrepid">intrepid</dt>
 <dd>{{site.data.glossary.intrepid}}</dd>

</dl>
```

If you squish your screen small enough, at a certain breakpoint this style reverts to the regular `dl` class.

Although I like the side-by-side view for shorter definitions, I found it problematic with longer definitions.

## FAQ layout

**Summary:** You can use an accordion-layout that takes advantage of Bootstrap styling. This is useful for an FAQ page.

If you want to use an FAQ format, use the syntax shown on the `faq.html` page. Rather than including code samples here (which are bulky with a lot of nested `div` tags), just look at the source in the `mydoc_faq.html` theme file.

Lorem ipsum dolor sit amet, consectetur adipiscing elit?

Curabitur eget leo at velit imperdiet varius. In eu ipsum vitae velit congue iaculis vitae at risus?

Aenean consequat lorem ut felis ullamcorper?

Lorem ipsum dolor sit amet, consectetur adipiscing elit?

Curabitur eget leo at velit imperdiet varius. In eu ipsum vitae velit congue iaculis vitae at risus?

Aenean consequat lorem ut felis ullamcorper?

Lorem ipsum dolor sit amet, consectetur adipiscing elit?

Curabitur eget leo at velit imperdiet varius. In eu ipsum vitae velit congue iaculis vitae at risus?

Aenean consequat lorem ut felis ullamcorper?

# Shuffle layout

**Summary:** This layout shows an example of a knowledge-base style navigation system, where there is no hierarchy, just groups of pages that have certain tags.

**Note:** The content on this page doesn't display well on PDF, but I included it anyway so you could see the problems this layout poses if you're including it in PDF.

[All](#)[Getting Started](#)[Formatting](#)[Publishing](#)[Content types](#)[Single Sourcing](#)[Special Layouts](#)

## Getting started

If you're getting started with Jekyll, see the links in this section. It will take you from the beginning level to comfortable.

- [Blain's World @ ACC \(page 3\)](#)
- [About the theme's author \(page 10\)](#)
- [About Ruby, Gems, Bundler, and other prerequisites \(page 16\)](#)
- [Install Jekyll on Mac \(page 24\)](#)
- [Pages \(page 32\)](#)
- [Posts \(page 39\)](#)
- [Release notes 5.0 \(page 14\)](#)

## Content types

This section lists different content types and how to work with them.

## Formatting

These topics get into formatting syntax, such as images and tables, that you'll use on each of your pages:

- [Tooltips \(page 82\)](#)
- [Alerts \(page 83\)](#)
- [Code samples \(page 103\)](#)
- [Glossary layout \(page 166\)](#)

- [Release notes 6.0 \(page 12\)](#)
- [Sidebar Navigation \(page 59\)](#)
- [Support \(page 11\)](#)
- [Supported features \(page 5\)](#)

- [Links \(page 105\)](#)
- [Icons \(page 91\)](#)
- [Images \(page 98\)](#)
- [Labels \(page 104\)](#)
- [Lists \(page 41\)](#)
- [Navtabs \(page 106\)](#)
- [Pages \(page 32\)](#)
- [Posts \(page 39\)](#)
- [Syntax highlighting \(page 114\)](#)
- [Tables \(page 110\)](#)
- [Workflow maps \(page 117\)](#)
- [YAML tutorial in the context of Jekyll \(page 62\)](#)

### Single Sourcing

These topics cover strategies for single\_sourcing. Single sourcing refers to strategies for re-using the same source in different outputs for different audiences or purposes.

- [Conditional logic \(page 45\)](#)
- [Content reuse \(page 50\)](#)
- [Excluding files \(page 0\)](#)
- [Generating PDFs \(page 128\)](#)

### Publishing

When you're building, publishing, and deploying your Jekyll site, you might find these topics helpful.

- [Build arguments \(page 124\)](#)
- [10. Configure the build scripts \(page 0\)](#)
- [Generating PDFs \(page 128\)](#)
- [Help APIs and UI tooltips \(page 141\)](#)

- [Help APIs and UI tooltips \(page 141\)](#)

- [iTerm profiles \(page 157\)](#)
- [Pushing builds to server \(page 159\)](#)
- [Search configuration \(page 153\)](#)
- [Themes \(page 127\)](#)

### Special Layouts

These pages highlight special layouts outside of the conventional page and TOC hierarchy.

- [FAQ layout \(page 169\)](#)
- [Glossary layout \(page 166\)](#)
- [Knowledge-base layout \(page 163\)](#)
- [Shuffle layout \(page 170\)](#)
- [Special layouts overview \(page 0\)](#)

**Note:** This was mostly an experiment to see if I could break away from the hierarchical TOC and provide a different way of arranging the content. However, this layout is somewhat problematic because it doesn't allow you to browse other navigation options on the side while viewing a topic.

# Troubleshooting

**Summary:** This page lists common errors and the steps needed to troubleshoot them.

## Issues building the site

### Address already in use

When you try to build the site, you get this error in iTerm:

```
jekyll 2.5.3 | Error: Address already in use - bind(2)
```

This happens if a server is already in use. To fix this, edit your config file and change the port to a unique number.

If the previous server wasn't shut down properly, you can kill the server process using these commands:

```
ps aux | grep jekyll
```

Find the PID (for example, it looks like "22298").

Then type `kill -9 22298` where "22298" is the PID.

Alternatively, type the following to stop all Jekyll servers:

```
kill -9 $(ps aux | grep '[j]ekyll' | awk '{print $2}')
```

### shell file not executable

If you run into permissions errors trying to run a shell script file (such as `mydoc_multibuild_web.sh`), you may need to change the file permissions to make the sh file executable. Browse to the directory containing the shell script and run the following:

```
chmod +x build_writer.sh
```

## shell file not runnable

If you're using a PC, rename your shell files with a .bat extension.

### "page 0" cross references in the PDF

If you see "page 0" cross-references in the PDF, the URL doesn't exist. Check to make sure you actually included this page in the build.

If it's not a page but rather a file, you need to add a `noCrossRef` class to the file so that your print stylesheet excludes the counter from it. Add

`class="noCrossRef"` as an attribute to the link. In the `css/printstyles.css` file, there is a style that should remove the counter from anchor elements with this class.

### The PDF is blank

Check the `prince-list.txt` file in the output to see if it contains links. If not, you have something wrong with the logic in the `prince-list.txt` file. Check the `conditions.html` file in your `_includes` to see if the audience specified in your configuration file aligns with the `buildAudience` in the `conditions.html` file

### Sidebar not appearing

If you build your site but the sidebar doesn't appear, check the following:

Look in your PDF config file and make sure you have a sidebar property, such as this:

```
pdf_sidebar: product2_sidebar
```

Make sure each TOC item has an output property that specifies web or pdf.

Understanding how the theme works can be helpful in troubleshooting. The `_includes/sidebar.html` file loops through the values in the `_data/sidebar.yml` file. There are `if` statements that check whether the conditions (as specified in the `conditions.html` file) are met. If the `sidebar.yml` item doesn't have the right output, then it won't get displayed in the sidebar. It would instead get skipped.



### Sidebar isn't collapsed

If the sidebar levels aren't collapsed, usually your JavaScript is broken somewhere. Open the JavaScript Console and look to see where the problem is. If one script breaks, then other scripts will break too, so troubleshooting it is a little tricky.

### Search isn't working

If the search isn't working, check the JSON validity in the search.json file in your output folder. Usually something is invalid. Identify the problematic line, fix the file, or put `search: exclude` in the frontmatter of the file to exclude it from search.