Jekyll theme for documentation — mydoc product

version 5.0

Last generated: March 18, 2016

Company
logo

Table of Contents

Handling reviews

Publishing

Special layouts

Troubleshooting

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 (http://www.komposta.net/article/navgoco) 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 (http://startbootstrap.com/template-overviews/modern-business/).

Getting started

To get started, see Get started (page 0).

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 feautres

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.
Markdown	Yes	You can author content using Markdown syntax. 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	I haven't done a translation project yet (just a pilot test). Here's the basic approach: Export the HTML pages and send them to a translation agency. Then create a new project for that language and insert the translated pages. Everything will be translated.

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 thousands of 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.
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).

FEATURES	SUPPORTED	NOTES
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 (http://talk.jekyllrb.com) is a great resource. So is Stack Overflow.
Blogging features	Yes	There is a simple blogging feature. This appears as "news" 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 plugins	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.

FEATURES	SUPPORTED	NOTES
Bootstrap integration	Yes	This theme is built on Bootstrap (http://getbootstrap.com/). 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.
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.

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 simplicy that makes Jekyll awesome. I recommend using WebStorm as your text editor.
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.
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.

FEATURES	SUPPORTED	NOTES
Templates for end-points	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.

Getting started

Summary: To get started with this theme, first make sure you have all the prerequisites in place; then build the theme following the sample build commands.

Getting up and running

To get up and running with this theme, make sure you can build a vanilla jekyll site first. See the Jekyll docs (http://jekyllrb.com/).

If you're in Windows, you might want to install Jekyll using Chocolately (https://www.google.com/search?q=install+jekyll+using+chocolately).

After ensuring you can run Jekyll on your machine, you can build this site using the usual Jekyll command: `jekyll serve`.

Configuring the theme

There are several products in this theme. Each product uses a different sidebar. This is the essence of what makes this theme unique – different sidebars for different product documentation. The idea is that when users are reading documentation for a specific product, the sidebar navigation should be specific to that product. The top navigation remains the same, because it allows users to navigate across products. But the sidebar navigation adapts to the product.

Where to store your documentation topics

Store your files for each product inside subfolders following the pattern shown in the theme. For example, product1, product2, etc. You can store your topics inside sub-subfolders to your heart's content. When Jekyll builds your site, it will pull the topics into the root directory and use the permalink for the URL.

Configuring the sidebar

Because each product uses a different sidebar, you'll need to set up your sidebars. There's a file inside _includes/custom called "sidebarconfigs.html". This file controls which sidebar gets associated with which product.

The sidebarconfigs.html file uses simple `if elsif` logic to set a variable that the sidebar.html file uses to read the sidebar data file. The code in sidebarconfigs.html looks like this:

```
{% if page.sidebar == "home_sidebar" %}
{% assign sidebar = site.data.sidebars.home_sidebar.entries %}

{% elsif page.sidebar == "product1_sidebar" %}
{% assign sidebar = site.data.sidebars.product1_sidebar.entrie
s %}

{% elsif page.sidebar == "product2_sidebar" %}
{% assign sidebar = site.data.sidebars.product2_sidebar.entrie
s %}

{% elsif page.sidebar == "mydoc_sidebar" %}
{% assign sidebar = site.data.sidebars.mydoc_sidebar.entries %}

{% elsif page.sidebar == "tags_sidebar" %}
{% assign sidebar = site.data.sidebars.tags_sidebar.entries %}

{% else %}
{% assign sidebar = site.data.sidebars.home_sidebar.entries %}
{% endif %}
```

In each page's frontmatter, you must specify the sidebar you want that page to use. Here's an example of the page frontmatter showing the sidebar property:

```
---
title: Alerts
tags: [formatting]
keywords: notes, tips, cautions, warnings, admonitions
last_updated: March 20, 2016
summary: "You can insert notes, tips, warnings, and important a
lerts in your content. These notes are stored as shortcodes mad
e available through the linksrefs.hmtl include."
sidebar: mydoc_sidebar
permalink: /mydoc_alerts/
---
```

The `sidebar: mydoc_sidebar` refers to the _data/sidebars/mydoc_sidebar.yml file.

If no sidebar assignment is found in the page frontmatter, the default sidebar (specified by the `else` statement) will be shown: `site.data.sidebars.home_sidebar.entries`.

Note that your sidebar can only have 2 levels. Given that each product has its own sidebar, this depth should be sufficient. Deeper nesting goes against usability recommendations.

Sidebar syntax

The sidebar data file uses a specific YAML syntax that you must follow. Follow the sample pattern shown:

```
- title: Overview
  output: web, pdf
  items:
  - title: Mydoc home
    url: /mydoc_landing_page/
    output: web
```

Each heading must contain a title and output property. Each item must contain a title, url, and output property.

The two outputs available are web and pdf. (Even if you aren't publishing PDF, you still need to specify `output: web`).

The YAML syntax depends on exact spacing, so make sure you follow the pattern shown in the sample sidebars. See my YAML tutorial (page 0) for more details about how YAML works.

To accommodate the title page and table of contents in PDF outputs, each product sidebar must list these pages before any other:

```
  - title:
    output: pdf
    type: frontmatter
    items:
    - title:
      url: /titlepage/
      output: pdf
      type: frontmatter
    - title:
      url: /tocpage/
      output: pdf
      type: frontmatter
```

Leave the output as `output: pdf` so that they don't appear in the web output.

Page frontmatter

When you write pages, include this same frontmatter in each page:

```
---
title: "Some title"
tags: [sample1, sample2]
keywords: keyword1, keyword2, keyword3
last_updated: Month day, year
summary: "optional summary here"
sidebar: sidebar name
permalink: /yoururl/
---
```

(If you're using Webstorm, you can set up a template to auto-populate this code when you create a new file.)

For titles, surrounding the title in quotes is optional, but if you have a colon in the title, you must surround the title with quotation marks.

Keywords get populated into the metadata of the page for SEO.

Tags must be defined in your _data/tags.yml list. You also need a corresponding tag file inside the tags folder that follows the same pattern as the other tag files shown in the tags folder. (Jekyll wont auto-create these tag files.) ```

If you don't want the mini-TOC to show on a page (such as for the homepage or landing pages), add `toc: none` in the frontmatter.

Configure the top navigation

The top navigation bar's menu items are set through the _data/topnav.yml file. Use the top navigation bar to provide links for navigating from one product to another, or to navigate to external resources.

For external URLs, use `external_url` in the item property, as shown in the example topnav.yml file. For internal links, use `url` as usual.

Note that the topnav has two sections: topnav and topnav_dropdowns. The topnav section contains single links, while the topnav_dropdowns section contains dropdown menus. The two sections are independent of each other.

Generating PDF

If you want to generate PDF, you'll need a license for Prince XML (http://www.princexml.com/). You will also need to install Prince (http://www.princexml.com/doc/installing/). You can generate PDFs by product (but not for every product on the site combined together into one massive PDF). Prince will work even without a license, but it will imprint a small Prince image on the first page.

Open up the css/printstyles.css file and customize the email address (`youremail@domain.com`) that is listed there. This email address appears in the bottom left footer of the PDF output. You'll also need to create a PDF configuration file following the examples shown in the pdfconfigs folder, and also customize some build scripts following the same pattern shown in the root: pdf-product1.sh

See the section on generating PDFs (page 0) for more details about setting the theme up for this output.

Blogs / News

For blog posts, create your markdown files in the _posts folder following the sample formats. Post file names always begin with the date (YYYY-MM-DD-title).

The news/news.html file displays the posts, and the news_archive.html file shows a yearly history of posts. In documentation, you might use the news to highlight product features outside of your documentation, or to provide release notes and other updates.

Markdown

This theme uses Kramdown markdown. Kramdown is similar to Github-flavored Markdown, except that when you have text that intercepts list items, the spacing of the intercepting text must align with the spacing of the first character after the space of a numbered list item.

Other instructions

For other details in working with the theme, see the various sections in the sidebar.

About the theme 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. My blog is here: http://idratherbewriting.com (http://idratherbewriting.com). I write several posts there a week. See my blog's about page (http://idratherbewriting.com/aboutme/) 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.

Support

> **Summary:** Contact me for any support issues.

Let me know about any bugs or other issues that you find. Just email me at
tomjohnson1492@gmail.com. You can also create issues directly within the
Github repository here (https://github.com/tomjohnson1492/jekyll-doc/issues).

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.

My preference is IntelliJ/WebStorm, since it will treat all files in your theme as belonging to a project. This allows you to easily search for instances of keywords, do find-and-replace operations, or do other actions that apply across the whole project.

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_*

These wildcards will exclude every match after the * .

Frontmatter

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

```
---
title: Alerts
tags: [formatting]
keywords: notes, tips, cautions, warnings, admonitions
last_updated: March 20, 2016
summary: "You can insert notes, tips, warnings, and important a
lerts in your content. These notes are stored as shortcodes mad
e available through the linksrefs.hmtl include."
sidebar: mydoc_sidebar
permalink: /mydoc_alerts/
---
```

Frontmatter is always formatted with three hyphens at the top a
nd bottom. Your frontmatter must have a `title` value. All the
other values are optional. If you omit them, the theme won't br
eak.

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 singl
e words, with underscores if needed (rather than spaces). Separ
ate them with commas. Enclose the whole list within brackets. A
lso, note that tags must be added to _data/tags_doc.yml to be
allowed entrance into the page. This prevents tags from becomin
g somewhat random and unstructured. You must create a tag page
for each one of your tags following the pattern shown in the ta
gs folder. (Tag pages aren't automatically created.)
keywords	Optional	Synonyms and other keywords for th
e page. This information gets stuffed into the page's metadata
to increase SEO. The user won't see the keywords, but if you se
arch for one of the keywords, it will be picked up by the searc
h engine.
last_updated	Optional	The date the page was last upd
ated. This information could helpful for readers trying to eval
uate how current and authoritative information is. If include
d, the last_updated date appears in the footer of the page in r
ather small font.|
summary	Optional	A 1-2 word sentence summarizing the
content on the page. This gets formatted into the summary secti
on in the page layout. Adding summaries is a key way to make yo
```

ur content more scannable by users (check out [Jakob Nielsen's
site](http://www.nngroup.com/articles/corporate-blogs-front-pag
e-structure/) for a great example of page summaries.) The only
drawback with summaries is that you can't use variables in the
m. |
| **permalink**| Required | This theme uses permalinks to facil
itate the linking. You specify the permalink want for the pag
e, and the \_site output will put the page into the root direct
ory when you publish. The page will appear inside a folder by t
he same name, with the actual page being index.html. Browsers w
ill automatically show the index.html file inside of any folde
r, so permalinks avoid the .html extension with file names. Per
malink names don't have to match your file names, but it might
be easier to keep them in sync. If you don't use permalinks, Je
kyll automatically uses the file name and folder path as the li
nk.|
| **datatable** | Optional | 'active'. If you add `datatable: a
ctive` in the frontmatter, scripts for the [jQuery Datatables p
lugin](https://www.datatables.net/) get included on the page. Y
ou can see the scripts that conditionally appear by looking in
the \_layouts/default.html page. |
| toc | Optional | If you specify `toc: none` in the frontmatte
r, the page won't have the table of contents that appears belo
w the title. The toc refers to the list of jump links below th
e 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 enclos
e the title's value in quotation marks.

## Saving pages as drafts

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

<div class="alert alert-success" role="alert"><i class="fa fa-c
heck-square-o"></i> <b>Tip: </b> You can create file templates
in WebStorm that have all your common frontmatter, such as all
possible tags, prepopulated. See <a href="/mydoc-pdf/mydoc_webs
torm_text_editor">WebStorm Text Editor</a> for details. </div>

## 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 not vice versa &mdash; if you use HTML (as your file extension), you can't insert Markdown content in the file.

Also, if you use HTML inside a Markdown file, you cannot use Markdown inside of HTML. But you can use HTML inside of Markdown.

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 flesh left in a Markdown file, all the tags inside those bookend HTML tags will render as HTML, regardless of their indentation.

## 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.

## 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 hae 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](http://idrath

```
erbewriting.com/2016/02/21/bug-with-kramdown-and-rouge-with-git
hub-pages/) for more details.

You can use standard Multimarkdown syntax for tables. You can a
lso use fenced code blocks with lexers specifying the type of c
ode. The configuration file shows the Markdown processor and ex
tensiosn:

```yaml
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: none` 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>`.

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.

Custom keyboard shortcuts

Some of the Jekyll syntax can be slow to create. Using a utility such as aText
(https://www.trankynam.com/atext/) can make creating content a lot of faster.

For example, with my aText configuration, when I type `jlink`, aText replaces it
with `<a href="{{ "/page" | prepend: site.baseurl }}">page</a>`.

You get aText from the App Store on a Mac for about $5. However, the Mac Store
version of aText won't work on Mac OSX El Capitan due to sandbox security
restrictions, so you need to download the app outside of the App Store to make it
work.

There are alternatives to aText, such as Typeitforme. But aText seems to work the
best. You can read more about aText on Lifehacker
(http://lifehacker.com/5843903/the-best-text-expansion-app-for-mac).

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 keywords: pages, authoring, exclusion, frontmatter summary: "This is some summary frontmatter for my sample post." sidebar: mydoc_sidebar permalink: /mydoc_pages/ tags: content_types —

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 _data/tags_doc.yml 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 tabs 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.
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 (http://www.nngroup.com/articles/corporate-blogs-front-page-structure/) for a great example of page summaries.) The only drawback with summaries is that you can't use variables in them.
permalink	Required	This theme uses permalinks to facilitate the linking. You specify the permalink want for the page, and the _site output will put the page into the root directory when you publish. The page will appear inside a folder by the same name, with the actual page being index.html. Browsers will automatically show the index.html file inside of any folder, so permalinks avoid the .html extension with file names. Permalink names don't have to match your file names, but it might be easier to keep them in sync.

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.

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**. (I would avoid the Multimarkdown plugin — it seemed to make all my dashes in frontmatter tags extend half way across the page.)

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**.

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
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.

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 (http://docs.shopify.com/themes/liquid-documentation/basics) 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 (http://docs.shopify.com/themes/liquid-documentation/basics/operators) for more details.

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

```
{% if page.audience contains "vegan" or page.audience == "veget
arian" %}
    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_amydoc_content_reuselpha
```

Then create a folder called _data_alpha.

For your beta configuratoin 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. See Jekyll's documentation (http://stackoverflow.com/questions/21976330/passing-parameters-to-inclusion-in-liquid-templates) for more information.

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 (http://ben.balter.com/2016/02/20/jekyll-collections/).

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 (http://jekyllrb.com/docs/collections/) 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 0) for details.

Video tutorial on collections

See this video tutorial on Jekyll.tips (http://jekyll.tips/jekyll-casts/introduction-to-collections/) for more details on collections.

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 (https://github.com/tefra/navgoco) 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 customsscripts.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 item.external_url %}
    <li><a href="{{item.external_url}}" target="_blank">{{subca
tegory.title}}</a></li>
    {% elsif page.url == item.url %}
```

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 == item.url %}
    <li class="active"><a href="{{item.url | prepend:
 ".."}}">{{item.title}}</a></li>
    {% else %}
    <li><a href="{{item.url | prepend: ".."}}">{{item.titl
e}}</a></li>
    {% endif %}
```

If the `page.url` matches the `item.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/. 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
      url: /mydoc_understand_sidebar/
      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 (https://github.com/tefra/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 ("**Y**AML **A**in't **M**arkup **L**anguage") 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/mydoc 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:

```
Husband's name: {{site.data.samplelist.name.husband}}

Wife's name: {{site.data.samplelist.name.wife}}
```

Notice that in order to access the data file, you use `site.data.samplelist`. `mydoc` is the folder, and `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:

```
**Feedback**
This is my feedback to you. Even if I include linebreaks here,
all of the linebreaks will be removed when the value is inserte
d.



**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.
```

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:

```
{% for item in site.data.samplelist.bikes %}
* {{item.title}}
{% endfor %}
```

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
 subitems:
    - location: US
    - location: Spain
    - location: France
```

Markdown + Liquid:

```
{% for item in site.data.samplelist.salesteams %}
<h3>{{item.title}}</h3>
<ul>
{% for entry in item.subitems %}
<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.subitems`.

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
    subitems:
      - page: Thing 1
      - page: Thing 2
      - page: Thing 3
  - title: Group 2
    subitems:
      - page: Piece 1
      - page: Piece 2
      - page: Piece 3
- title: Group 3
    subitems:
      - 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.subitems %}
<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 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:

```
{% for sec in site.data.samplelist.mypages %}
{% if sec.audience == "writers" %}
* {{sec.url}}
{% endif %}
{% endfor %}
```

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:

```
{% for sec in site.data.samplelist.mypages %}
{% if sec.audience == "writers" and sec.product == "gizmo" %}
* {{sec.url}}
{% endif %}
{% endfor %}
```

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
(http://www.tournemille.com/blog/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: 2.0 Release Notes
permalink: /release_notes_2_0/
tags: [formatting, single_sourcing]
---
```

Tags overview

> ❶ **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.

> ❶ **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/
    sidebar: tags_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_" | prepend: site.baseurl | append: t
ag }}" class="btn btn-default navbar-btn cursorNorm" rol
e="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 page (page 0) 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 | prepend: '..'}}">{{page.titl
e}}</a></li>
{% endif %}
{% endfor %}
{% endfor %}
</ul>
```

Here's how that code renders:

Getting started pages:

- [Introduction (page 0)](#)

- [About the theme author (page 16)](#)

- [Getting started (page 10)](#)

- [Pages (page 18)](#)

- [Posts (page 24)](#)

- [Sidebar Navigation (page 38)](#)

- [Support (page 17)](#)

- [Supported features (page 4)](#)

- [Troubleshooting (page 136)](#)

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 | prepend: '..'}}">{{page.titl
e}}</a></li>
{% endif %}
{% endfor %}
{% endfor %}
</ul>
```

Here's how that code renders:

Getting started pages:

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 0) 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.

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="bt
n btn-primary dropdown-toggle">Series Demo <span class="care
t"></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}}</l
i>
            {% else %}
            <li>
                <a href="{{p.url | prepend: '..'}}">{{p.weigh
t}}. {{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 | prepend: '..'}}"><button type="button" c
lass="btn btn-primary">Next: {{p.weight}}  {{p.title}}</butto
n></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 0)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 0).

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.</a>
```

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, just reference the appropriate value stored in the alerts.yml file as described in the following sections.

Alerts

You can insert an alert by using any of the following code.

ALERT	CODE
note	{{site.data.alerts.note}} your note {{site.data.alerts.end}}
tip	{{site.data.alerts.tip}} your tip {{site.data.alerts.end}}
warning	{{site.data.alerts.warning}} your warning {{site.data.alerts.end}}
important	{{site.data.alerts.important}} your important info {{site.data.alerts.end}}

The following demonstrate the formatting associated with each alert.

> ☑ **Tip:** Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.

> ❶ **Note:** Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.

> ⚠ **Important:** Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.

> ❗ **Warning:** Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.

Callouts

In contrast to the alerts, the callouts don't have a pre-coded bold-formatted preface such as note or tip. You just add one (if desired) in the callout text itself.

CALLOUT	CODE
callout_default	{{site.data.alerts.callout_default}} your callout_default content {{site.data.alerts.end}}
callout_primary	{{site.data.alerts.callout_primary}} your callout_primary content {{site.data.alerts.end}}
callout_success	{{site.data.alerts.callout_success}} your callout_success content {{site.data.alerts.end}}
callout_warning	{{site.data.alerts.callout_warning}} your callout_warning content {{site.data.alerts.end}}
callout_info	{{site.data.alerts.callout_info}} your callout_info content {{site.data.alerts.end}}

The following demonstrate the formatting for each callout.

> **callout_danger**: Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard

dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.

callout_default: Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.

calloutprimary: Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.

calloutsuccess: Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.

calloutinfo: Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.

calloutwarning: Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.

Blast a warning to users

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 defaut) will show this message.

Using Markdown inside of notes

You can't use Markdown formatting inside alerts. This is because the alerts leverage HTML, and you can't use Markdown inside of HTML tags. It's very easy to forget this, which is why I recommend using HTML formatting for links in every case. This way you're less likely to forget to switch into HTML mode when you're writing content in a tip. You must remember, however, to avoid Markdown with code and bold formatting inside of notes.

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).

See Font Awesome icons available

Go to the Font Awesome library (http://fortawesome.github.io/Font-Awesome/icons/) 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:
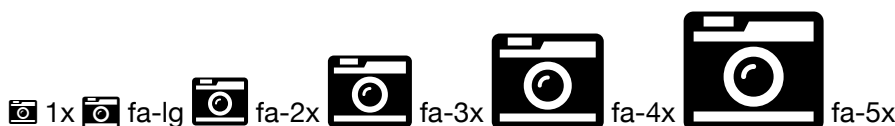
With Font Awesome, you always use the `i` tag with the appropriate class. You also implement `fa` as a base class first. You can use font awesome icons inside other elements. Here I'm using a Font Awesome class inside a Bootstrap alert:

```
<div class="alert alert-danger" role="alert"><i class="fa fa-ex
clamation-circle"></i> <b>Warning: </b>This is a special warnin
g 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="f
a 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 cl
ass="fa fa-warning"></i> <b>Important: </b>'
warning: '<div class="alert alert-danger" role="alert"><i clas
s="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.

```
{{site.data.alerts.note}} Add your note here. {{site.data.alert
s.end}}
```

```
{{site.data.alerts.tip}} Add your tip here. {{site.data.alert
s.end}}
```

```
{{site.data.alerts.important}} Add your important info here.
{{site.data.alerts.end}}
```

```
{{site.data.alerts.warning}} Add your warning here. {{site.dat
a.alerts.end}}
```

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-d
ownload fa-lg"></i> This is a special tip about some file to do
wnload....</div>
```

And the result:

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

Grab the right class name from the Font Awesome library
(http://fortawesome.github.io/Font-Awesome/icons/) 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
(http://getbootstrap.com/components/#glyphicons) to see the icons available.

Although the Glyphicon Halflings library doesn't provide the scalable classes like
Font Awesome, there's a StackOverflow trick
(http://stackoverflow.com/questions/24960201/how-do-i-make-glyphicons-bigger-change-
size)
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:

Glypicons 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-hidde
n="true"></span>
  <b>Error:</b> Enter a valid email address
</div>
```

> ❶ **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 messag
e. This is a special info message. This is a special info messa
ge. This is a special info message. This is a special info mess
age. This is a special info message. This is a special info mes
sage. This is a special info message. </div>
```

> ❷ 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. 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 0).

Images

Summary: You embed images using traditional HTML or Markdown syntax for images. Unlike pages, you can store images in subfolders (in this theme). This is because when pages reference the images, the references are always as subpaths, never requiring the reference to move up directories.

You embed an image the same way you embed other files or assets: you put the file into a folder, and then link to that file.

Put images inside the `images` folder in your root directory. You can create subdirectories inside this directory. Although you could use Markdown syntax for images, the HTML syntax is probably easier:

```
<img title="my sample page" src="{{ "/images/jekyll.png" | prep
end: site.baseurl }}" />
```

And the result:

Here's the same Markdown syntax:

```
![My sample page]({{ "/images/jekyll.png" | prepend: site.baseu
rl }})
```

And the result:

☑ **Tip:** I recommend storing this format into a shortcut editor such as aText. This way when you want to insert an image, just type something like jimg and the shortcut editor will automatically type the code.

Figure captions

If you want to add a figure caption, you can do so using standard `figure` HTML tags:

```
<figure><img title="my sample page" src="{{ "/images/jekyll.pn
g" | prepend: site.baseurl }}" /><figcaption>Your caption</figc
aption></figure>
```
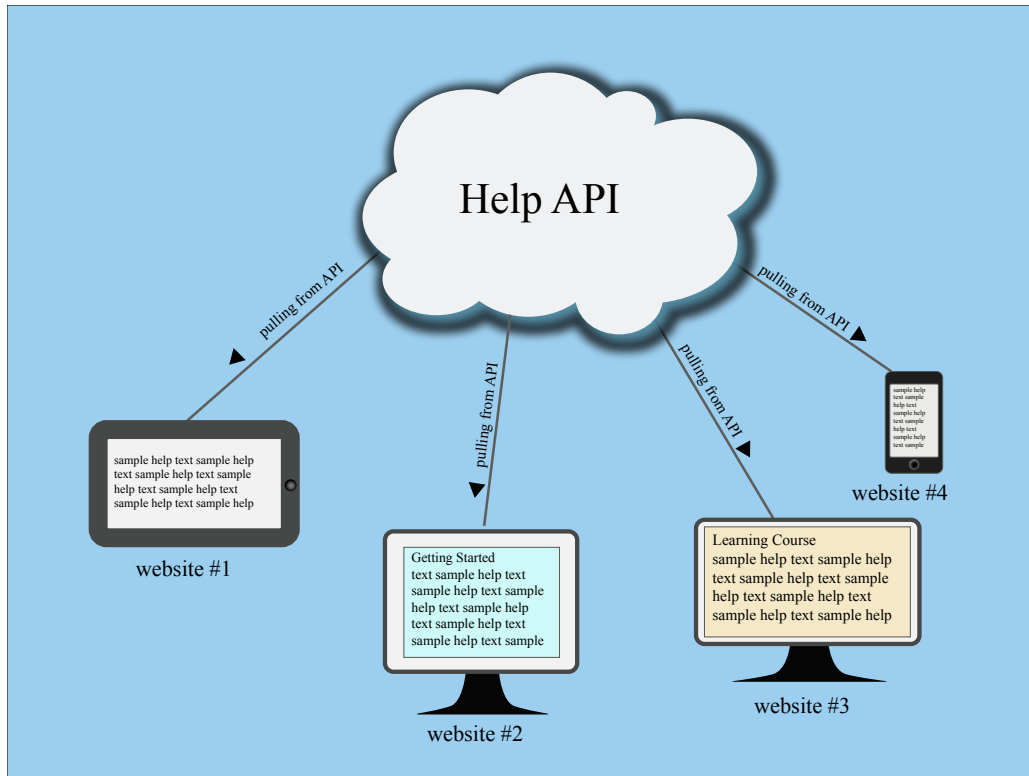
Here's the result:

Your caption

</figure>

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:

```
<img src="/mydoc-pdf/images/helpapi.svg" style="max-width: 600p
x;" />
```

Here's the result:

SVG images will expand to the size of their container, so you have to specify it here. The previous syntax isn't well supported in IE, so you would be better off using the `object` element like this:

```
<div style="max-width:600px;"><object type="image/svg+xml" dat
a="{{ "/images/helpapi.svg" | prepend: site.baseurl }}">Your br
owser does not support SVG</object>
</div>
```

Here's the same code with `figure` elements:

```
<div style="max-width:600px;"><figure><object type="image/svg+x
ml" data="{{ "/images/helpapi.svg" | prepend: site.baseurl
}}">Your browser does not support SVG</object><figcaption>This
is your caption</figcaption></figure>
</div>
```

And the result:
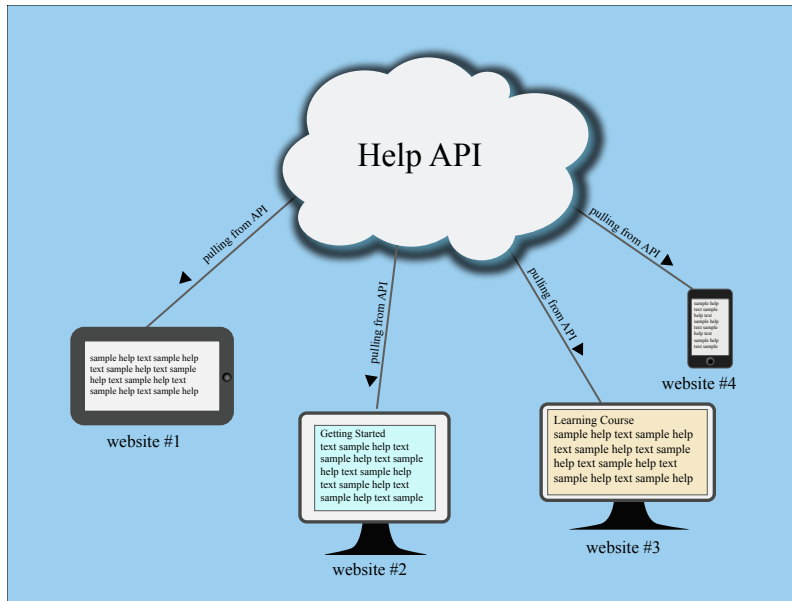
This is your caption

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.
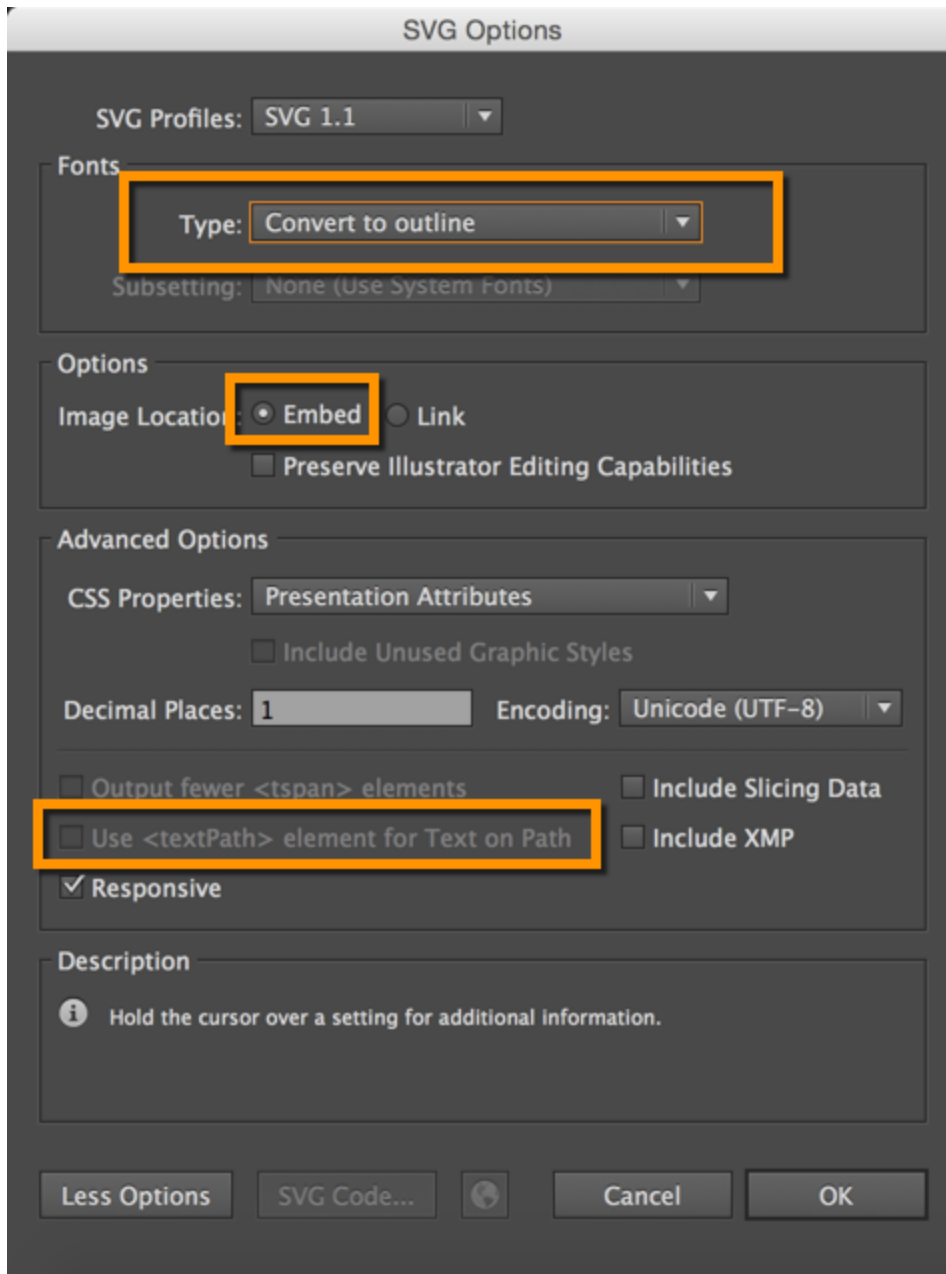
Labels

> **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. Note that this approach is susceptible to errors and broken links, so check your outputs for broken links.

Create an external link

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

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

If you need to use HTML, use the normal syntax:

```
<a href="http://google.com">Google</a>
```

Linking to internal pages

When linking to internal pages, you can use this same syntax:

```
[Sample](/mydoc-pdf/page)
```

OR

```
<a href="/mydoc-pdf/page">page</a>
```

I find that using the HTML formatting is easiest. Store the code in a shortcut in aText to populate it easily.

Avoiding broken links

In general, avoid broken links and outdated titles in links by doing the following:

- Where possible, avoid using the exact titles in link names. For example, if you write, see the Links (page 78) page, this title is likely to become more outdated than if you were to write, learn how to manage links (page 78).

- Use a broken link checker on your site output to see if links are broken.

- Generate a PDF, since the PDF tends to highlight broken links more forcefully.

Other methods for managing links

You can also adopt an indirect-reference systems for managing links. This involves storing the link text in YAML syntax.

If you want to try this method, look in the root directory. The urls.txt file contains the same code as the table of contents (but without the conditional qualifiers), duplicated for each of the sidebars. The code iterates through every page listed in the table of contents sidebars (as well as the top navigation menus) and creates an output that looks like this for each link:

```
mydoc_introduction:
  title: "Introduction"
  url: "mydoc_introduction"
  link: "<a href='/mydoc_introduction/'>Introduction</a>"
```

From the site output folder (in _site), open urls.txt and observe that it is properly populated (blank spaces between entries doesn't matter). Then manually copy the contents from the urls.txt and insert it into the _data/urls.yml file in your project folder.

Because the urls.txt is produced from the table of contents, you ensure that the same titles and URLs used in your table of contents and top navigation will also be used in your inline links.

To create a link in a topic, just reference the appropriate value in the urls.yml file, like this:

```
{{site.data.urls.mydoc_introduction.link}}
```

This will insert the following into your topic:

```
<a href='/mydoc_getting_started/'>Getting started</a>
```

You don't need to worry whether you can use Markdown syntax when inserting a link this way, because the insertion is HTML.

To insert a link in the context of a phrase, you can use this syntax:

```
After downloading the theme, you can [get started in building t
he theme]({{site.data.urls.mydoc_getting_started.url}}).
```

This leverages Markdown syntax. If you're in an HTML file or section, use this:

```
<p>After downloading the theme, you can <a href="{{site.data.ur
ls.mydoc_getting_started.url}}">get started in building the the
me</a>.</p>
```

Note that the `url` value accesses the URL for the page only, whereas `link` gets the title and url in a link format.

You shouldn't have to copy the contents from the urls.txt file into your YAML data source too often — only when you're creating new pages.

By using this approach, you're less likely to end up with broken links.

> ☑ **Tip:** To avoid having to remember this long syntax, use a text macro program like aText (https://itunes.apple.com/us/app/atext/id488566438?mt=12).

Always make sure your TOC page is accurate

You should treat your sidebar data files (in /_data/sidebars) with a lot of care. Every time you add a page to your site, make sure it's listed in your sidebar file (or in your top navigation). If you don't have pages listed in your sidebar file, they won't be included in the urls.txt file, and as your site grows, it will be harder to recognize pages that are absent from the TOC.

Because all the pages are stored in the root directory, the list of files can grow really long. I typically find pages by navigating to the page in the preview server, copying the page name (e.g., mydoc_hyperlinks), and then pressing **Shift + Shift** in WebStorm to locate the page.

This is the only sane way to locate your pages when you have hundreds of pages in your root directory. If the page isn't listed in your TOC, it will be difficult to navigate to it and find it.

Checking for broken links

Another way to ensure you don't have any broken links in your output is to generate a PDF (page 0). When you generate a PDF, look for the following two problems in the output:

- page 0

- see .

Both instances indicate a broken link. The "page 0" indicates that Prince XML couldn't find the page that the link points to, and so it can't create a cross reference. This may be because the page doesn't exist, or because the anchor is pointing to a missing location.

If you see "see ." it means that the reference (for example, `{{mylink...}}` doesn't actually refer to anything. As a result, it's simply blank in the output.

> **ⓘ Note:** To keep Prince XML from trying to insert a cross reference into a link, add `class="noCrossRef"` to the link.

Navtabs

Summary: Navtabs provide a tab-based navagation 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, nibh in u ltrices mattis, felis ipsum venenatis metus, vel vehicula libero mauris a enim. Sed placerat est ac lectus vestibulum tempor. Quisque 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 (http://stackoverflow.com/questions/10523433/how-do-i-keep-the-current-tab-active-with-twitter-bootstrap-after-a-page-reload)](http://stackoverflow.com/questions/10523433/how-do-i-keep-the-current-tab-active-with-twitter-bootstrap-after-a-page-reload).

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:

```
Column 1 | Column 2
--------|----------
cell 1a | cell 1b
cell 2a | cell 2b
```

This renders to the following:

COLUMN 1	COLUMN 2
cell 1a	cell 1b
cell 2a	cell 2b

jQuery datables

You also have the option of using a jQuery datatable (https://www.datatables.net/), which gives you some more options. If you want to use a jQuery datatable, then add `datatable: active` in a page's frontmatter. This will load the right jQuery datatable scripts for the table on that page only (rather than loading the scripts on every page of the site.)

Also, you need to add this script to trigger the jQuery table on your page:

```
<script>
$(document).ready(function(){

    $('table.display').DataTable( {
        paging: true,
        stateSave: true,
        searching: true
    }
        );
});
</script>
```

The available options for the datable are described in the datatable documentation (https://www.datatables.net/manual/options), which is excellent.

Additionally, you must add a class of `display` to your tables. (You can change the class, but then you'll need to change the trigger above from `table.display` to whatever class you want to you. You might have different triggers with different options for different tables.)

Since Markdown doesn't allow you to add classes to tables, you'll need to use HTML for any datatables. Here's an example:

```
<table id="sampleTable" class="display">
    <thead>
        <tr>
            <th>Parameter</th>
            <th>Description</th>
            <th>Type</th>
            <th>Default Value</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>Parameter 1</td>
            <td>Sample description
            </td>
            <td>Sample type</td>
            <td>Sample default value</td>
        </tr>
        <tr>
            <td>Parameter 2</td>
            <td>Sample description
            </td>
            <td>Sample type</td>
            <td>Sample default value</td>
        </tr>
     <tr>
        <td>Parameter 3</td>
        <td>Sample description
        </td>
        <td>Sample type</td>
        <td>Sample default value</td>
    </tr>
        <tr>
            <td>Parameter 4</td>
            <td>Sample description
            </td>
            <td>Sample type</td>
            <td>Sample default value</td>
        </tr>
    </tbody>
</table>
```

This renders to the following:

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 in-sides 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 (https://www.datatables.net/manual/options) 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.

> ❶ **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.

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:

```
```ruby
  def foo
    puts 'foo'
  end
```
```

This looks as follows:

```
  def foo
    puts 'foo'
  end
```

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 ruby %}
  def foo
    puts 'foo'
  end
{% endhighlight %}
```

It renders the same:

```
def foo
  puts 'foo'
end
```

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

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:

```
{% unless jekyll.environment == "production" %}

    {% if site.github_editme_path %}

    <a target="_blank" href="https://github.com/{{site.github_editme_path}}{% unless page.url contains "html" %}{{page.url | replace: '.html', '.md'}}{% endunless %}{% if page.url contains "html" %}{{page.url }}{% endif %}" class="btn btn-default githubEditButton" role="button"><i class="fa fa-github fa-lg"></i> Edit me</a>
    {% endif %}

    {% endunless %}
```

This code is only active if you're publishing in a development environment, which is the default.

To activate the production environment, add the production environment flag (http://jekyllrb.com/docs/configuration/) in your build command:

```
JEKYLL_ENV=production jekyll serve
```

In your configuration file, edit the value for `github_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': tomjohnson1492/documentation-theme-jekyll/edit/reviews. Here "tomjohnson1492" is my github account name. The repo name is "documentation-theme-jekyll". The "reviews" name is the branch.

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.

> ❶ **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.

> ❶ **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 (http://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:

```
jekyll 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:

```
jekyll 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:

```
jekyll 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 0) 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.

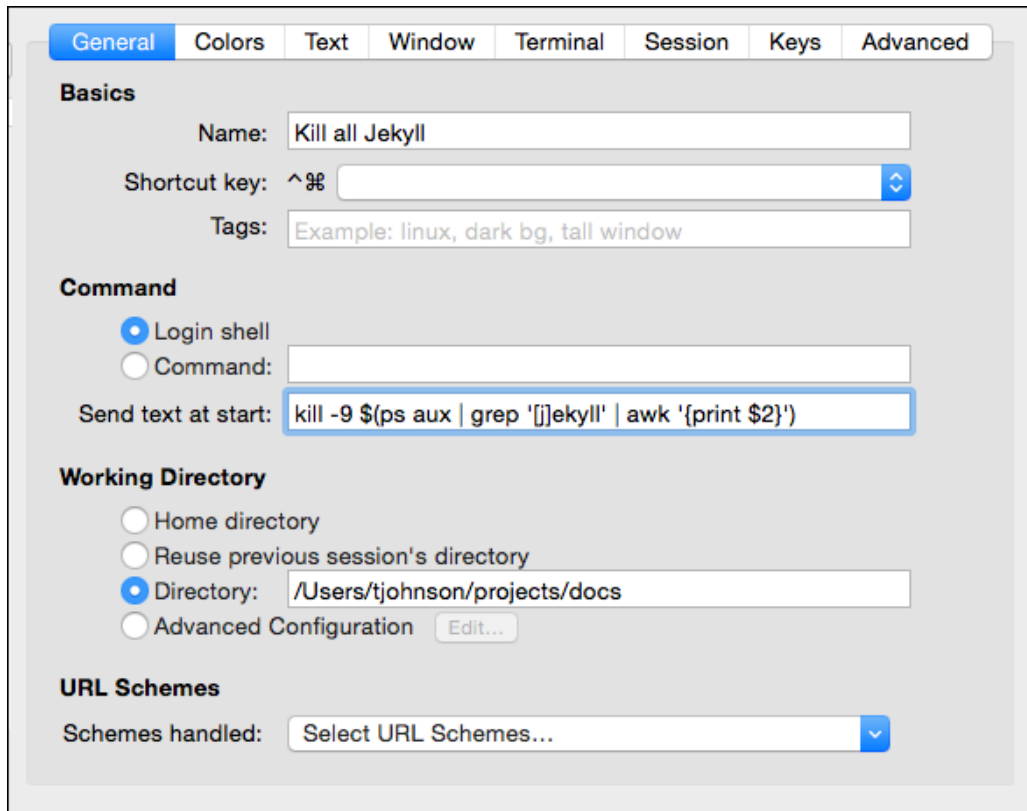You can kill the server process using these commands:

```
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:

| General | Colors | Text | Window | Terminal | Session | Keys | Advanced |

Basics

Name: Kill all Jekyll

Shortcut 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...

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.

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="/mydoc-pdf/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.

Check page title consistency

> **Summary:** The title checker page helps ensure that the titles in your pages match the titles in your TOC.

The theme has a file called title-checker.html. This file will iterate through all the pages listed in the sidebar navigation and top navigation, and compare the navigation titles against the page titles based on matching URLs. If there are inconsistencies in the titles, they get noted on the title-checker.html page.

To run the link checker, just build or serve your project, and go to /title-checker in your browser (such as Chrome). If there are inconsistencies, they will be noted on the page.

Note that in order for the title-checker file to run correctly, it has to detect a match between the URL listed in the sidebar or top navigation with the URL for the page (based on the file name). If you have the wrong URL, it won't tell you if the page titles match. Therefore you should always click through all the topics in your navigation to make sure the URLs are accurate.

> ❶ **Note:** If your page titles have your product name in parentheses, but your sidebar doesn't have the product name in parentheses, this title-checker tool is going to return a lot of mismatches. This is one limitation of the code right now.

Note also that you must manually configure your sidebar file in the first line of the code, and then repeat the same chunk of code for each sidebar. Right now the code doesn't automatically iterate over every sidebar file. It's somewhat of a manual configuration process there.

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.

The basic approach is to generate a list of all pages that need to be added to the PDF, and then add leverage Prince to package them up into a PDF.

It may seem like the setup is somewhat cumbersome, but it doesn't take long. 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

1. Set up Prince

Download and install Prince (http://www.princexml.com/doc/installing/).

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 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
```

> ⓘ **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` toggle 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.

3. Make sure your sidebar_doc.yml file has a titlepage.html and tocpage.html

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
  items:
  - title:
    url: /titlepage/
    output: pdf
    type: frontmatter
  - title:
    url: /tocpage/
    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.

> ❶ **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.typ
e == "frontmatter" %}frontmatter{% elsif page.type == "first_pa
ge" %}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: March 18, 2016";
    });
</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(doctitle);
```

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 the PDF script

Open the pdf-mydocf.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_my
doc_pdf.yml;
echo "done";

echo "Building the PDF ...";
prince --javascript --input-list=_site/pdfconfigs/prince-list.t
xt -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 a download button for the PDF

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

```
<a target="_blank" class="noCrossRef" href="/pdf/mydoc.pdf"><bu
tton type="button" class="btn btn-default" aria-label="Left Ali
gn"><span class="glyphicon glyphicon-download-alt" aria-hidde
n="true"></span> PDF Download</button></a>
```

Here's what that looks like:

[⬇ PDF Download]

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:

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

For more detail about using `unless` in conditional logic, see . 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 Bootrap, 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 tooltip demo.

In this demo, the popovers pull in and display content from the information in a mydoc_tooltips_source.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 _data > mydoc 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:

```
---
id: basketball
product: mydoc
---

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

You need to create a separate page 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 "mydoc_tooltips_source.json." (You can use whatever name you want.) Add the following to your JSON file:

```
---
layout: none
search: exclude
---
{
"entries":
[
{% for page in site.tooltips %}
{% if page.product == "mydoc" %}
{
"id"    : "{{ page.id }}",
"body": "{{ page.content | strip_newlines | replace: '\',
'\\\\' | replace: '"', '\\"' }}"
} {% unless forloop.last %},{% endunless %}
{% endif %}
{% endfor %}
]
}
```

Change "mydoc" to the product name you used in each of the tooltip files. The template here will only include content in the JSON file if it meets the `product` attribute requirements. We need this `if` statement to prevent tooltips from other products from being included in the JSON file.

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

You can also view the same JSON file here: mydoc_tooltips_source.json.

You can add different fields depending on how you want the JSON to be structured. Here we just have to fields: `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
> (https://google-styleguide.googlecode.com/svn/trunk/jsoncstyleguide.xml). These
> best practices can help you keep your JSON file valid.

You can store your mydoc_tooltips_source.json file anywhere you want, but to me it make sense to store it inside a tooltips folder for your specific project. This way it will automatically be excluded from other projects that are already excluding that project directory.

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, though 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 mydoc/tooltips/mydoc_tooltips_source.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://idratherbetellingstories.com/wp-content/apidemos/, I uploaded a file called ".htaccess" with the preceding code. You can view it [here](http://idratherbetellingstories.com/wp-content/apidemos/mydoc_tooltips_source.json).

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://idratherbetellingstories.com/wp-content/apidemos/mydoc_tooltips_source.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 (http://enable-cors.org/server.html) 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 a page with the ID of `basketball`:

```javascript
<script type="text/javascript">
$(document).ready(function(){

var url = "mydoc_tooltips_source.json";


$.get( url, function( data ) {

        $.each(data.entries, function(i, page) {
            if (page.id == "basketball") {
            $( "#basketball" ).attr( "data-content", page.body
);
            }

        });
    });

});
</script>
```

View the Tooltip Demo for a demo.

The `url` here 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.

> ❶ **Warning: Note:** Make sure your JSON file is valid. Otherwise, this method won't work. I use the JSON Formatter extension for Chrome (https://chrome.google.com/webstore/detail/json-formatter/ bcjindcccaagfpapjjmafapmmgkkhgoa?hl=en) . 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 (http://jsonformatter.curiousconcept.com/). 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 (http://getbootstrap.com/javascript/#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 <span class="glyphicon glyphicon-info-sign" id="basketball" data-toggle="popover"></span></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/bo
otstrap/3.3.2/css/bootstrap.min.css">
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.1
1.2/jquery.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/j
s/bootstrap.min.js"></script>

<script type="text/javascript">
$(document).ready(function(){
    $('[data-toggle="popover"]').popover({
        placement : 'right',
        trigger: 'hover',
        html: true
    });
```

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. Take a look at that code if you want to change what fields are included.

The search is a simple search that looks at content in pages. It looks at titles, summaries, keywords, tags, and bodies.

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 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-file-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, go directly to the search.json file in your browser, and then copy the content. Go to a JSON validator (http://jsonlint.com/) and paste in the content. Look for the line causing trouble. Edit the file to either exclude it from search or fix the syntax so that it doesn't invalidate the JSON.

The search.json file already tries to strip out content that would otherwise make the JSON invalid:

```
      "body": "{{ page.content | strip_html | strip_newlines |
 replace: '\', '\\\\' | replace: '"', '\\"' | replace: '^t',
 '    ' }}",
```

Note that the last replace, `| replace: '^t', ' '` , looks for any tab character and replaces it with four spaces. Yes, an innocent little tab character invalidates JSON. Geez. If you run into other problematic formatting, you can use regex expressions to find and replace the content. See Regular Expressions (http://www.ultraedit.com/support/tutorials_power_tips/ultraedit/regular_expressions.html) 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.)

iTerm profiles

Summary: 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're single sourcing multiple outputs. It can be a hassle to fire up each one of these outputs using the build files containing the shell scripts. Instead, it's easier to configure iTerm with profiles that initiate the scripts.
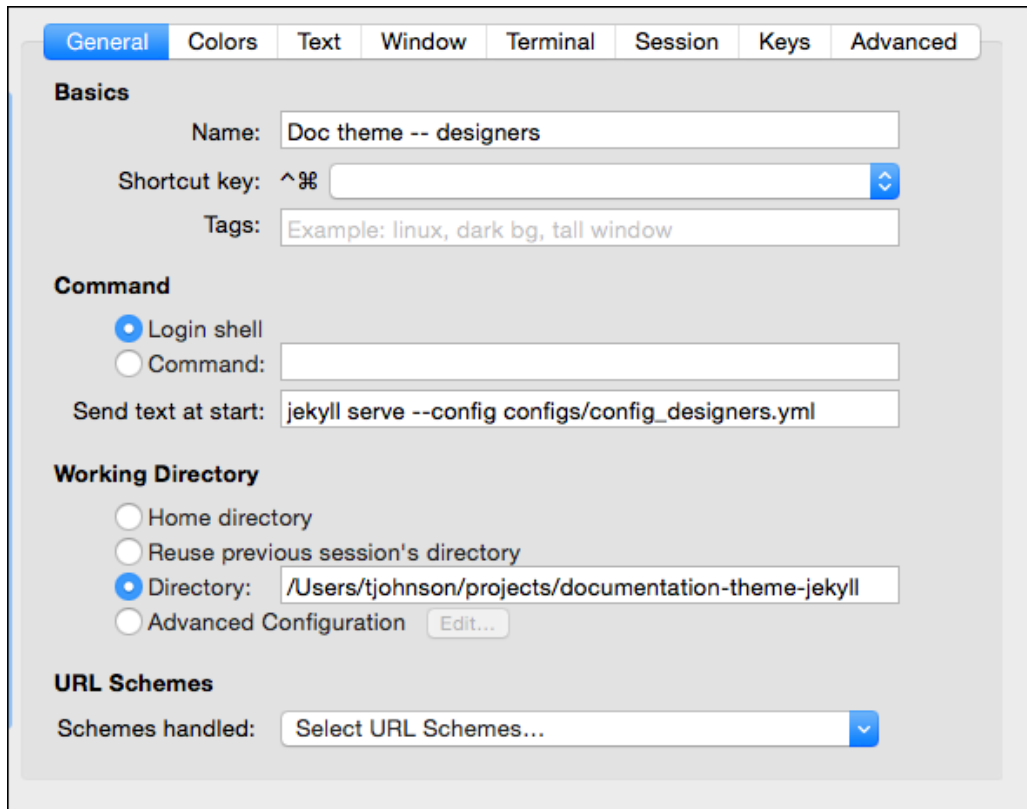
Set up profiles

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 serve --config configs/config_designers.yml
   ```

   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:

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/documentation-theme-jekyl
l-builds/mydoc_writers s3://[aws path]documentation-theme-jekyl
l/mydoc_writers --recursive

#aws s3 cp ~/users/tjohnson/projects/documentation-theme-jekyl
l-builds/mydoc_designers s3://[aws path]/documentation-theme-je
kyll/mydoc_designers --recursive
```

The first path 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/documentation-theme-jekyll-buil
ds/mydoc_writers name@domain:/var/www/html/documentation-them
e-jekyll/mydoc_writers
```

Similar to the above, the first path is the local location; the second path is the destination.

Getting around the password prompts in SCP

You can publish your docs via SSH through a Terminal window or more likely, via a shell script that you simply execute as part of the publishing process. However, you will be prompted for your password with each file transfer unless you configure passwordless SSH.

The basic process for setting up password less SSH is to create a key on your own machine that you also transfer to the remote machine. When you use the SCP command, the remote machine checks that you have the authorized key and allows access without a password prompt.

To remove the password prompts when connecting to servers via SSH:

1. On your local machine, go to your .ssh directory:

```
cd ~/.ssh
```

Note that any directory that starts with a dot, like .ssh, is hidden. You can view hidden folders by enabling them on your Mac. See this help topic (http://ianlunn.co.uk/articles/quickly-showhide-hidden-files-mac-os-x-mavericks/). Additionally, when you look at the files in a directory, use ls -a instead of just ls to view the hidden files.

If you don't have an .ssh directory, create one with `mkdir .ssh`.

Create a new key inside your .ssh directory:

```
ssh-keygen -t rsa
```

Press Enter. When prompted about "Enter file in which to save the key …", press Enter again.

This will create a file called id_rsa.pub (the key) and id_rsa (your identification) in this .ssh folder.

When prompted for a passphrase for the key, just leave it empty and press Enter twice. You should see something like this:

tjohnson-mbpr13:.ssh tjohnson$ ssh-keygen -t rsa Generating public/private rsa key pair. Enter passphrase (empty for no passphrase): Enter same passphrase again: Your identification has been saved in /Users/tjohnson/.ssh/id_rsa. Your public key has been saved in /Users/tjohnson/.ssh/id_rsa.pub. The key fingerprint is: 9a:8f:b5:495:39:78:t5:dc:19:d6:29:66:02:e8:02:a0 tjohnson@tjohnson-mbpr13.local The key's randomart image is:

```
+--[ RSA 2048]----+
|.               |
|+               |
|E               |
|o.   .          |
|.. = o S         |
|.&^  + 7i = o       |
|      = B .      |
|      o 0 +      |
|       *.o       |
+----------------+
```

Icon As you can see, RSA draws a picture for you. Take a screenshot of the picture, print it out, and put it up on your fridge.

Open up another terminal window (in iTerm, open another tab), and SSH in to your remote server:

```
ssh <your_username>@remoteserver.com
```

Change `<your_username>` to your actual username, such as tjohnson.

When you connect, you'll be prompted for your password.

When you connect, by default you are routed to the personal folder on the directory. For example, `/home/remoteserver/<your_username>`. To see this directory, type `pwd`.

Create a new directory called .ssh on remoteserver.com server inside the `/home/remoteserver/<your_username>` directory.

```
mkdir -p .ssh
```

You can ensure that it's there with this command:

```
ls -a
```

Without the -a, the hidden directory won't be shown.

Open another Terminal window and browse to /Users//.ssh on your local machine.

```
cd ~/.ssh
```

Copy the id_rsa.pub from the /.ssh directory on your local machine to the /home/remoteserver//.ssh directory on the remoteserver server:

```
scp id_rsa.pub <your-username>@yourserver.com:/home/remoteserve
r/<your-username>/.ssh
```

Switch back into your terminal window that is connected to remoteserver.com, change directory to the .ssh directory, and rename the file from id_rsa.pub to `authorized_keys` (without any file extension):

```
mv id_rsa.pub authorized_keys
```

Change the file permissions to 700:

```
chmod 700 authorized_keys
```

Now you should be able to SSH onto remoteserver without any password prompts.

Open another terminal (which is not already SSH'd into remoteserver.com) and try the following:

```
ssh <your_username>@remoteserver.com
```

If successful, you shouldn't be prompted for a password.

Now that you can connect without password prompts, you can use the scp scripts to transfer files to the server without password prompts. For example:

```
scp -r ../doc_outputs/mydoc/writers <your-username>@remoteserve
r:/var/www/html/
```

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 enlarged to a large size. You can also add captions below each icon.

Getting Started

Navigation

single_sourcing

Publishing

Special layouts

Formatting

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: ```html 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.title}} (page 0)

{% endif %} {% endfor %} {% endfor %}

``` Getting started pages:

- About the theme author (page 16)

- Getting started (page 10)

- Introduction (page 0)

- Pages (page 18)

- Posts (page 24)

- Sidebar Navigation (page 38)

- Support (page 17)

- Supported features (page 4)

- Troubleshooting (page 136)

# 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:

```
<dl class="dl">

<dt id="fractious">fractious</dt>
<dd>Like a little mischevious child, full of annoying and const
ant trouble.</dd>

<dt id="gratuitous">gratuitous</dt>
<dd>Something that is unwarranted and uncouth, like the social
equivalent of a flagrant foul.</dd>

<dt id="haughty">haughty</dt>
<dd>Proud and flaunting it. Holding your head high up like a sn
ooty, too-good-for-everything rich person.</dd>

<dt id="gratuitous">gratuitous</dt>
<dd>Something that is unwarranted and uncouth, like the social
equivalent of a flagrant foul.</dd>

<dt id="impertinent">impertinent</dt>
<dd>Someone acting rude and insensitive to others.</dd>

<dt id="intrepid">intrepid</dt>
<dd>Brave and courageous especially in a difficult, dangerous s
ituation.</dd>

</dl>
```

Here's what that looks like:

**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.

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.

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.

# 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}')
```

Build not entirely finishing

If your build doesn't entirely finish on the command line, check to see if you have a space after a comma when using multiple configuration files, like this:

```
jekyll serve --config config_base.yml, config_designer.yml
```

Remove the space after the comma, and the build will finish executing:

```
jekyll serve --config config_base.yml,config_designer.yml
```

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
```

Pygments not installed

The config file requires pygments for the highlighter. You must download and install Pygments (http://pygments.org/download/), which requires Python, in order to use this syntax highlighter. If you don't want to bother with Pygments, open the configuration file and change `pygments` to `rouge` .

"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-file-list.txt file in the output to see if it contains links. If not, you have something wrong with the logic in the prince-file-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 _includes/custom/sidebarconfigs.html and make sure the conditional values there match up with the values declared in the configuration file. Specifically, you need to make sure you've declared a value for project, product, platform, and version.

If you don't have any values for these properties, you still need to keep them in your configuration file. Just put something like `all` as the value.

> **❶ Note:** This theme is designed for single sourcing. If you're only building one site, you can remove these values from the _includes/sidebar.html file and _data/sidebar.yml files.

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 has the right product, platform, audience, and version, then it gets displayed in the sidebar. If not, it get skipped.

Sidebar heading level not opening

In your _data/sidebar.yml file, you must also include the correct parameters (platform, product, audience version) for each heading. If an item contains something that should be displayed, the attributes for the heading should be listed.

Without any attributes on heading levels, you could end up with scenarios where a section is entirely designed for one output but appears in every output regardless.

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.

Adding all project dependencies

Summary:

You want to be sure that you have all the required gems and other utilities on your computer to make the project run. Jekyll runs on Ruby, and there are various plugins for Ruby that enable different functionality. These Ruby plugins are referred to as gems, and you install the gems you need for your projects.

To manage the various gems and their versions needed for your project, you can use a package manager called Bundler. Many projects will have a gemfile in their project that lists the gems required for the project. You then run Bundler in order to automatically install the required gems and any dependencies for those gems on your machine.

RubyGems

Make sure you have RubyGems. This should be installed by default on Mac.

Open Terminal and type `which gem`. You should get a response indicating the location of Rubygems, such as `/Users/MacBookPro/.rvm/rubies/ruby-2.2.1/bin/gem`.

If you need to install Rubygems, see Download RubyGems (https://rubygems.org/pages/download).

Install Bundler

Bundler (http://bundler.io/) is a package manager for RubyGems.

You install Bundler by using the gem command with RubyGems:

```
gem install bundler
```

If you're prompted to which 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.

If you get a permissions error when trying to install Bundler, use Homebrew to install a Ruby package manager called rbenv.

Install Homebrew:

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

Use brew to install rbenv:

```
brew install rbenv
```

Initialize rbenv:

```
rbenv init
```

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

Install Bundler:

```
gem install bundler
```

Open the gemfile:

```
open gemfile
```

You should see some gems listed. If you don't have a gemfile, your project may not need any gems, or those gems may not be managed at the project level but rather directly installed manually. You can create a gemfile by typing `bundle init`.

Your gemfile might look like this:

```
# A sample Gemfile
source "https://rubygems.org"

# gem "rails"
gem 'github-pages'
gem 'pygments.rb'
gem 'redcarpet'
```

Use Bundler to install the needed gems:

```
bundle install
```

Finally, you can run the following to make sure the installed gems get associated or initialized with your project:

```
bundle exec jekyll serve
```

Ruby permissions errors

While trying to install a missing gem, you get an error message that says,

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

This most likely happens with El Capitan on the Mac.

As long as you have brew installed (see the previous section), run the following:

```
brew update
brew install ruby
```

Close your terminal, and then restart a fresh session.

Now run the gem you're trying to install, such as the following:

```
gem install kramdown
```