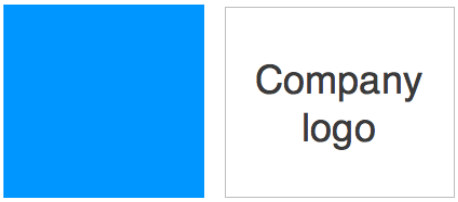
Jekyll theme for documentation — writers

version 4.0

Last generated: December 02, 2015



© 2015 Your company. This is a boilerplate copyright statement... All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Overview

Introduction 1

Supported features 3

Get started

1. Build the default project 8

2. Add a new project 12

3. Decide on your project's attributes 15

4. Set the configuration options 17

5. Customize the conditions file 27

6. Configure the sidebar 29

7. Configure the top navigation 34

8. Customize the URL generator 36

9. Set up Prince XML 37

10. Configure the build scripts 38

Authoring

Pages 44

WebStorm Text Editor 50

Conditional logic 53

Content reuse 58

Collections 60

Navigation

Sidebar navigation 62

Tags 65

Series 71

Formatting

Tooltips 74

Alerts 75

Icons 79
Images 85
Labels 88
Links 89
Navtabs 94
Video embeds 97
Tables 101
Syntax highlighting 105

Handling reviews

Commenting on files 107

Publishing

Build arguments 108
Themes 111
Link validation 112
Check page title consistency 114
Generating PDFs 115
Excluding files 127
Help APIs and UI tooltips 130
Search configuration 142
iTerm profiles 144
Pushing builds to server 146
Getting around the password prompts in SCP 147

Special layouts

Knowledge-base layout 151
Scroll layout 155
Shuffle layout 161
FAQ layout 165
Glossary layout 166

Troubleshooting

Troubleshooting 169

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 set up to push out documentation for multiple projects all from the same source. You can also share content across projects.

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

- Bootstrap framework
- Sidebar for table of contents
- Top navigation bar with drop-down menus
- PDF generation (through Prince XML utility)
- Build scripts to automate the workflow
- Notes, tips, and warning information notes
- A nifty system for creating links to different pages
- Tags for alternative nativation
- Content sharing across projects
- Emphasis on pages, not posts
- Relative (rather than absolute) link structure, so you can push the outputs anywhere and easily view them

I'm using this theme for my documentation projects, building about 20 different outputs for various products, versions, languages, and audiences from the same set of files. This single sourcing requirement has influenced how I constructed this theme.

For more discussion about the available features, see Supported features (page 3).

To get started, see these three topics:

1. Getting started with this theme (page 8)
2. Setting configuration options (page 17)
3. Adding new projects (page 12)

If you would like to download this help file as a PDF, you can do so here. The PDF is comprehensive of all the content in the online help.

PDF Download

The PDF contains a timestamp in the header indicating when it was last generated.

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. The following table shows what is supported in Jekyll and this theme.

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.
Translation	Yes	I haven't done a translation project yet (just a pilot test). Here's the basic approach: Export the 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.
PDF	Yes	You can generate PDFs from your Jekyll site. This theme uses Prince XML (costs $495) to do the PDF conversion task. You basically set up a page that uses Liquid logic to get all the pages you want, and then you use PrinceXML (not part of Jekyll) to convert that page into a PDF.

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.
Multichannel output	Yes	This term can mean a number of things, but let's say you have 10 different sites you want to generate from the same source. Maybe you have 7 different versions of your product, and 3 different locations. You can assemble your Jekyll site with various configurations, variants, and more. Jekyll actually does all of this quite well. Just specify a different config file for each unique build.
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).
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.

FEATURES	SUPPORTED	NOTES
Blogging features	No	This theme just uses pages, not posts. I may integrate in post features in the future, but the theme really wasn't designed with posts in mind. If you want a post version of the site, you can clone my blog theme (https://github.com/tomjohnson1492/tomjohnson1492.github.io), which is highly similar in that it's based on Bootstrap, but it uses posts to drive most of the features. I wanted to keep the project files simple.
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, but ...	As noted in the previous point, 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.
Versioning	Yes, but...	Jekyll doesn't version your files. You upload your files to a version control system such as Git. Your files are versioned there.
PC platform	Yes, but ...	Jekyll isn't officially supported on Windows, and since I'm on a Mac, I haven't tried using Jekyll on Windows. See this page in Jekyllrb help (http://jekyllrb.com/docs/windows/) for details about installing and running Jekyll on a Windows machine. A couple of Windows users who have contacted me have been unsuccessful in installing Jekyll on Windows, so beware. In the configuration files, use `rouge` instead of `pygments` (which is Python-based) to avoid conflicts.

FEATURES	SUPPORTED	NOTES
jQuery plug-ins	Yes	You can use any jQuery plugins you and other JavaScript, CMS, or templating tools. However, note that if you use Ruby plugins, you can't directly host the source files on Github Pages because Github Pages doesn't allow Ruby plugins. Instead, you can just push your output to any web server. If you're not planning to use Github Pages, there are no restrictions on any plugins of any sort. Jekyll makes it super easy to integrate every kind of plugin imaginable. This theme doesn't actually use any plugins, so you can publish on Github if you want.
Bootstrap integration	Yes	This theme is built on Bootstrap (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.
Relative links	Yes	This theme is built entirely with relative links, which means you can easily move the files from one folder to the next and it will still display. You don't need to view the site on a web server either -- you can view it locally just clicking the files. This relative link structure facilitates scenarios where you need to archive versions of content or move the files from one directory (a test directory) to another (such as a production directory).
Themes	Yes	You can have different themes for different outputs. If you know CSS, theming both the web and print outputs is pretty easy.

FEATURES	SUPPORTED	NOTES
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.

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. Because this theme is set up for single sourcing projects, it doesn't follow the same pattern as most Jekyll projects (which have just a _config.yml file in the root directory).

Getting Started ▾

Before you start installing the theme, make sure you have all of these prerequisites in place.

- **Mac computer**. If you have a Windows machine, make sure you can get a vanilla Jekyll site working before proceeding. You'll probably need Ruby and Ruby Dev Kit installed first. Also note that the shell scripts (.sh files) in this theme for automating the builds work only on a Mac. To run them on Windows, you need to convert them to BAT.
- **Ruby** (https://www.ruby-lang.org/en/). On a Mac, this should already be installed. Open your Terminal and type `which ruby` to confirm.
- **Rubygems** (https://rubygems.org/pages/download). This is a package manager for Ruby. Type `which gem` to confirm.
- **Text editor**: My recommendations is WebStorm (or IntelliJ). You can use another text editor. However, there are certain shortcuts and efficiencies in WebStorm (such as using Find and Replace across the project, or Markdown syntax highlighting) that I'll be noting in this documentation.

Before you start customizing the theme, make sure you can build the theme with the default content and settings first.

1. Download the theme from the documentation-theme-jekyll Github repository (https://github.com/tomjohnson1492/documentation-theme-jekyll) and unzip it into your ~username/projects folder.

You can either download the theme files directly by clicking the **Download Zip** button on the right of the repo, or use git to clone the repository to your local machine.

2. After downloading the theme, note some unique aspects of the file structure:
    - Although there's a _config.yml file in the root directory, it's there only so that Github Pages will build the theme. Because the theme is set up for single sourcing, there's a separate configuration file for each unique output you're building.
    - All the configuration files are stored in the configs directory. Each configuration file has a different preview port. If you want, you can build and preview all your outputs simultaneously in different preview servers.
    - Each configuration file specifies a different project and potentially a different audience, product, platform, and version. By setting unique values for these properties in the includes/custom/conditions.html file, you determine how the sidebar and top navigation get constructed.
    - A variety of shell scripts (.sh files) in the project's root directory automate the building and publishing of both the web and PDF files. After you configure the scripts, you can execute all the scripts through the master shell script, mydoc_all.sh.
    - "mydoc" is the name of the documentation project. You can leave the mydoc content in the theme. It won't affect the other projects you add to the theme.

> ☑ **Tip:** The main goal of this theme is to enable single sourcing. With single sourcing, you build multiple outputs from the same source, with somewhat different content in each output based on the particular product, platform, version, and audience. You don't have to use this theme for single sourcing, but most tech writing projects require this. If you're not a technical writer or not creating documentation, this theme is probably not for you. It doesn't have any post features coded, just pages.

There are four configuration files in this project: config_writers.yml and config_designers.yml as well as their PDF equivalents. The idea is that there's an output specific to writers, and an output specific to designers.

In reality, both of these outputs are pretty much the same. I mainly incorporated two outputs here mainly to demonstrate how the single sourcing works.

3. Unless you're planning to publish on Github Pages, you can delete the Gemfile. The Gemfile is only in this project to allow publishing on Github Pages.

   The theme is not using a Gemfile to manage project dependencies. Although theoretically the Gemfile should make things easier, I've found that it tends to give users more errors than they need. Add to this the incompatibility of Github Pages with Jekyll 3.0 and the Gemfile becomes even more problematic.

4. Install the Jekyll (https://rubygems.org/gems/jekyll), redcarpet (https://rubygems.org/gems/redcarpet), and pygments (https://rubygems.org/gems/pygments.rb) gems.

   These gems are the only ones the project uses. Go to the Rubygems site (https://rubygems.org) for each of these gems (based on the links above). In the right column, click the "INSTALL" command and paste the copied command into your terminal. If your computer gives you permissions errors, add `sudo` before the command.

5. In your terminal, browse to the documentation-theme-jekyll folder that you downloaded.

6. Build the writer's output:

   ```
   jekyll serve -config configs/mydoc/config_writers.yml
   ```

   The `--config` parameter specifies the location of the configuration file to be used in the build. The configuration file itself contains the destination location for where the site gets built.

   Open a new tab in your browser and preview the site at the preview URL shown.

7. Press **Ctrl+C** in Terminal to shut down the writer's preview server.

8. Build the designers output:

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

   Open a new tab in your browser and preview the site at the preview URL shown. Notice how the themes differ (designers is blue, writers is green).

9. Press **Ctrl+C** in Terminal to shut down the designer's preview server.
10. Build both themes by running the following command:

```
. mydoc_3_multibuild_web.sh
```

The themes build in the `../doc_outputs/mydoc/mydoc_designers` and `../doc_outputs/mydoc/mydoc_writers` folders. Browse to these directories to view the output.

Open the writers and designers folders and click the index.html file. The themes should launch and appear similar to their appearance in the preview folder. This is because the themes are built using a relative link structure, so you can move the theme to any folder you want without breaking the links.

If the theme builds both outputs successfully, great. You can move on to the other sections. If you run into errors building the themes, solve them before moving on. See Troubleshooting (page 169) for more information.

> ☑ **Tip:** You can set up profiles in iTerm to initiate all your builds with one selection. See iTerm profiles (page 144) for details.

More information about building the PDF versions is provided in Generating PDFs (page 115).

If you have questions, contact me at tomjohnson1492@gmail.com. My regular site is idratherbewriting.com (http://idratherbewriting.com). I'm eager to make these installation instructions as clear as possible, so please let me know if there are areas of confusion that need clarifying.

Next: 2. Add a new project (page 12)

Summary: You add a new project essentially by duplicating all the mydoc project files in the _data, _includes, configs, and other folders. You can add as many projects as you want in this theme.

Getting Started

The theme shows two build outputs: one for designers, and one for writers. The dual outputs is an example of the single sourcing nature of the theme. The designers output is comprehensive, whereas the writers output is a subset of the information. However, the outputs are mostly the same. I just created the separate output to demonstrate how the single sourcing aspect works.

You can add as many documentation projects as you want to the same Jekyll project. Some doc projects have multiple outputs, as is the case with the designers and writers outputs for the mydoc project.

Follow these steps to add additional projects.

Important: In these instructions, I'll assume your project's name is "acme." Replace "acme" with the real name of your project.

1. Copy and customize the mydoc folder in _data

Inside the _data folder, copy the mydoc folder and its contents. Rename it to acme, and then rename each of the YML files inside the folder with the acme prefix.

The files in data control how the side and top nav bar get populated. Here is also where URLs, definitions, and other settings are stored.

2. Copy and customize the mydoc folder in configs

In the configs folder, copy the mydoc folder and its contents. Rename it to acme, and then rename each of the config_ files to the outputs you need for your acme project.

In this theme, each output requires a separate config file. If you have 10 audiences and you want separate sites for each, then then you'll have 10 config files in this directory.

More details about customizing the settings in the configuration files will be explained later. For now you're just duplicating the necessary project files for your new project.

3. Create a includes folder

In the _includes/custom directory, add a new folder there called "acme." This folder should sit parallel to the mydoc folder. This is where you can store includes for your project.

4. Add an acme folder in the root directory

In the root directory, add a folder for your pages called acme (similar to the mydoc folder). Include two subfolders inside acme: files and images.

Inside the mydoc folder, copy the home.md file and add it to the acme folder. (With most Jekyll projects, they open up on the index.html file in the root directory. However, because the pages for each project are stored in subfolders, it was necessary to create a redirect from the index page to the home.md page.)

This acme directory is where you'll store all your pages.

Note that you cannot create subfolders in this acme directory. All of your pages have to be flat in this directory. This is because the references to the resources (stylesheets, javascript, etc.) are relative, and creating additional directory levels will break the relative paths.

5. Copy and customize the mydoc shell scripts in the root directory

In the root directory, duplicate the shell scripts and rename the prefix to "acme_". The following files are the shell scripts that need to be duplicated:

- mydoc_1_multiserve_pdf.sh
- mydoc_2_multibuild_pdf.sh
- mydoc_3_multibuild_web.sh

- mydoc_4_publish.sh
- mydoc_all.sh

6. Copy the URL generator text file

In the root directory, copy urls_mydoc.txt and duplicate it. Change the suffix to urls_acme.txt.

> ☑ **Tip:** In this step, you're just duplicating project files. In later steps, you'll actually customize all of the settings.

Next: 3. Decide on your project's attributes (page 15)

Summary: Each project has attributes that define the audience, platform, product, version, and output. These attributes are used in generating the outputs. The attributes function as filtering conditions that determine what content gets included in the navigation.

Getting Started

Before you can customize your project's settings, you have to make some decisions about the following:

- `audience`
- `platform`
- `product`
- `version`
- `output` (web, pdf)

Every project uses a value for these settings, so even if the attribute doesn't apply to your project, you will need to put some value for `audience`, `platform`, `product`, and `version` (a value such as `all` will work fine).

The `audience`, `platform`, `product`, and `version` settings derive from the same filtering attributes as in DITA. You can usually create any kind of filtered output by combining these attributes in different ways.

For example, you might have different product lines (lite versus pro), different versions (1.0 versus 2.0), different platforms (such as Java versus C++), and different audiences (administrators versus analysts) and so on. You'll need to know the values you want to use for each attribute in order to configure the project successfully to build the different outputs.

If you aren't sure of your outputs, just put `all` for the `audience`, `platform`, `product`, and `version`. For the `output` value, the options are fixed to either `web` or `pdf` (or both — `web, pdf` — separated by a comma).

Next: 4. Set the configuration options (page 17)

Summary: The configuration file contains important settings for your project. Some of the values you set here affect the display and functionality of the theme — especially the product, platform, audience, and version.

Getting Started

The configuration file serves important functions with single sourcing. For each site output, you create a unique configuration file for that output.

The configuration file contains most of the settings and other details unique to that site output, such as variables, titles, output directories, build folders, and more.

You can define arbitrary key-value pairs in the configuration file, and then you can access them through `site.yourkey`, where `yourkey` is the name of the key.

However, some of the options you set in the configuration file determine theme settings. These options are required for this theme to work. The required settings are defined in the following tables.

The values in the following tables are used to control different aspects of the theme and are not arbitrary key-value pairs. As you set up your project, enter the appropriate values for each of these keys in the configuration file.

If you're unsure how or where the project setting affects the theme, just search for the project setting in the theme (for example, `site.sidebar_version`) and you'll see the files involved.

The order of the settings doesn't matter.

FIELD	REQUIRED?	DESCRIPTION
project	Required	A unique name for the project. The _includes/custom/conditions.html file will use this project name to determine what sidebar and top nav data files to use. Make this value unique. Note that the project name also determines what conditions are set in the conditions.html file. It's critical that the project name you specify in the configuration file matches the project names in the conditions.html file. Otherwise, the conditions.html file won't be able to set the right variables needed for single sourcing. (Admittedly, the settings for these attributes are somewhat duplicated between the conditions.html and configuration file.)
audience	Required	The audience for the output. This value is also set in the _includes/custom/conditions.html file. Each entry in _data/sidebar_doc.yml and _data/topnav_doc.yml needs to have an audience attribute that matches the correct audience value in order for the sidebar or topnav item to be included.
platform	Required	The platform for the output. The same matching logic applies as with audience.
product	Required	The product for the output. The same matching logic applies as with audience.
version	Required	The version for the output. The same matching logic applies as with audience.

FIELD	REQUIRED?	DESCRIPTION
destination	Required	The folder where the site is built. If you put this into your same folder as your other files, Jekyll may start building and rebuilding in an infinite loop because it detects more files in the project folder. Make sure you specify a folder outside your project folder, by using `../` or by specifying the absolute path. The recommended output folder is `../doc_outputs`. The PDF configuration files will look in that directory for the outputs needed to build the PDF outputs.
sidebar_tagline	Optional	Appears above the sidebar. Usually you put some term related to the site specific build, such as the audience name. In the sample theme files, the taglines are "writers" and "designers." Keep these short — there's not much room. Six or seven letters is perfect.
sidebar_version	Optional	Appears below the sidebar_tagline in a smaller font, usually specifying the version of the documentation. In the sample theme files, the version is "4.0."
topnav_title	Required	Appears next to the home icon in the top nav bar. In the sample theme files, the topnav_title is "Jekyll Documentation Theme."

FIELD	REQUIRED?	DESCRIPTION
homepage_title	Required	You set the title for your homepage via this setting. This is because multiple projects are all using the same index.md as their homepage. Because index.md has `homepage: true` in the frontmatter, the "page" layout will use the `homepage_title` property from the configuration file instead of the traditional title in the frontmatter. In the sample theme files, the homepage title is "Jekyll Documentation Theme — writers" or "Jekyll Documentation Theme — designers."
site_title	Required	Appears in the webpage title area (on the browser tab, not in the page viewing area). In the sample theme files, the site title is rendered as `{{ page.title }}{% endif %}
port	Required	The port used in the preview mode. This is only for the live preview and doesn't affect the published output. If you serve multiple outputs simultaneously, the port must be unique.
feedback_email	Required	Gets configured as the email address in the Send Feedback button in the top navigation bar.
disqus_shortname	Optional	The Disqus site shortname, which is used for comments. If you don't want comment forms via disqus, leave this blank or omit it altogether and Disqus won't appear.
markdown	Required	The processor used to convert Markdown to HTML. This is a Jekyll-specific setting. Use `redcarpet`. Another option is `kramdown`. However, my examples will follow redcarpet.

FIELD	REQUIRED?	DESCRIPTION
redcarpet	Required	Extensions used with redcarpet. You can read more about the Red Carpet extensions here (https://github.com/vmg/redcarpet).
highlighter	Required	The syntax highlighter used. Use `rouge` because it has fewer dependencies on your operating system (it doesn't require Python). However, you can also use `pygments`. If so, you may need need to install Pygments (http://pygments.org/download/).
exclude	Optional	A list of files and directories that you want excluded from the build. By default, all the content in your project is included in the output. If you don't want to include a file or directory, list it here. It's helpful to name your files with a prefix such as product_audience_filename.md, so that you can exclude using wildcards such as "product*" or product_audience*". For more information about excluding files, see Excluding files (page 127).
defaults	Optional	Here you can set default values for frontmatter based on the content type (page, post, or collection).

FIELD	REQUIRED?	DESCRIPTION
collections	Optional	Any specific collections (custom content types that extend beyond pages or posts) that you want to define. This theme defines a collection called tooltips. You access this collection by using site.tooltips instead of site.pages or site.posts. Put the tooltip content types inside a folder in your project called _tooltips. Tooltips are useful for creating UI content. For more information about creating tooltips for UI text, see Help APIs and UI tooltips (page 130).
output	Optional	Boolean. Whether this build is `web` or `pdf` . This setting allows you to run conditions in your content such as `{% if site.output == pdf %}` do this... `{% endif %}` . Limit the options to just `web` or `pdf` for this setting.
github_editme_path	Optional	A path to configure the Github Edit Me button. Put the path to the branch on Github where you want to edit the theme. Here's a sample: `tomjohnson1492/documentation-theme-jekyll/edit/reviews` . In this case, "reviews" is the name of the branch where I want people to make edits. I can then merge the "reviews" branch with the "gh-pages" branch (which is the default branch). See the "page" layout (inside the _layouts folder) for how this path gets inserted into the rest of the HTML.
company_name	Optional	Used in the footer to brand your site.

FIELD	REQUIRED?	DESCRIPTION
footer_image_location	Optional	The image used in the footer to brand your site. Store this image in the common_images folder so that it's not excluded by a particular project. Example: ../common_images/company_logo.png
theme_file	Optional	The theme used for the output. Currently there are two options: theme-green.css or theme-blue.css. These themes cover both web and PDF output. The themes have the same style and layout. They only differ in the accent color for the top nav bar, buttons, hyperlinks, and other small details.
pdf_file_name	Optional	The name of the PDF file generated by Prince. This is helpful for the code on the home.md page that allows users to download a PDF of the material. If you have 5 different PDFs, you don't want to use `if` statements to render different PDF buttons. Instead, this theme uses the same PDF code but swaps out the PDF file name with a variable here.

In this theme, all the configuration files are listed in the configs directory. There are some build scripts in the root directory that reference the configuration files in this configs folder.

There's also a _config.yml file in the root directory. This is simply copied from the configs directory and used to accommodate publishing with Github Pages.

The PDF configuration files build on all the settings in the web configuration files, but they add a few more options.

When you build the PDF output (such as for the writers output), the command will look like this:

```
jekyll serve --detach --config configs/config_writers.yml,confi
gs/config_writers_pdf.yml
```

First Jekyll will read the config_writers.yml file, and then Jekyll will read the config_writers_pdf.yml file. Values from both configuration files will be used, but the later configuration file (on the right) will overwrite any values set in the previous configuration file (on the left).

(Previously people running Windows reported problems with cascading the configuration files like this. If you're on Windows, for PDF outputs, you may need to combine the settings from the web configuration file into the PDF configuration file.)

More detail about generating PDFs is provided in Generating PDFs (page 115), but the configuration settings used for the PDFs are described here.

The process for creating PDFs relies on two steps:

1. First you build a printer-friendly web version of the content.
2. Then you run PrinceXML to get all the printer-friendly web pages and package them into a PDF.

Thus, you actually build a web version for the PDF first before generating the PDF.

The following table describes the settings in the PDF configuration file.

FIELD	REQUIRED?	DESCRIPTION
destination	Where the PDF web version should be served so that Prince XML can find it. By default, this is in `../mydoc_designers-pdf` , which is just one level above where your project is.	
url	The URL where the files can be viewed. This is `http://127.0.0.1:4002` in the sample theme files for the designers output. Prince XML requires a URL to access the file.	

FIELD	REQUIRED?	DESCRIPTION
baseurl	The subdirectory after the url where the content is stored. In the sample theme files for the designers output, this is `/designers`.	
port	The port required by the preview server.	
output	`web` or `pdf`. This setting allows you to construct conditional statements in your content to check whether output is web or pdf. This setting can help you filter out content that doesn't fit well into a PDF (such as dynamic web elements). In particular, the Prince XML script conflicts with any JavaScript on the page, so you want to filter out the JavaScript from the PDF-friendly HTML output that Prince consumes.	
print_title	The title for the PDF. In the sample theme files for designers output, the print title is "Jekyll Documentation Theme for Designers"	
print_subtitle	The subtitle for the PDF. In the sample theme files, the subtitle is "version 4.0."	
defaults	See the sample settings in the config_designers_pdf.yml file. The only difference between this file and config_designers.yml is that the layout used for pages is `page_print` instead of `page`. The `page_print` layout also used `head_print` instead of `head`. This layout strips out components such as the sidebar and top navigation. It also leverages printstyles.css and includes some JavaScript for Prince XML.	

Summary: The conditions file is included in various parts of the theme. Its purpose is to set attributes as variables that affect how the theme is constructed. The settings in this file are essential for single sourcing.

Getting Started

The conditions file is a critical file that sets certain variables used in constructing the theme. You already set some of these values in the configuration file, but you need to duplicate some of the settings here. In this file, the settings are variable assignments.

This file is used as include in certain files. When used as an include, it sets variables that are used to configure your theme. Because you're single sourcing your Jekyll content, you need this file.

In the _includes/custom directory, open the conditions.html file. Duplicate one of the project settings blocks like this:

```
{% if site.project == "mydoc_writers" %}
{% assign audience = "writers" %}
{% assign sidebar = site.data.mydoc.mydoc_sidebar.entries %}
{% assign topnav = site.data.mydoc.mydoc_topnav.topnav %}
{% assign topnav_dropdowns = site.data.mydoc.mydoc_topnav.topna
v_dropdowns %}
{% assign version = "all" %}
{% assign product = "all" %}
{% assign platform = "all" %}
{% assign projectTags = site.data.mydoc.mydoc_tags.allowed-tag
s %}
{% assign projectFolder = "mydoc" %}
{% endif %}
```

You need to duplicate this block for each output you have.

Once you've duplicated the block, make a few customizations:

- In each place that "mydoc" appears, change "mydoc" to "acme".
- Use the same attributes for project, audience, version, product, and platform that you used in your configuration file. (If you don't have a specific attribute value that you need, just put "all".) The values here have to exactly match those in the configuration file.

☑ **Tip:** If you want to create signposts in the code as shown in the conditions.html file, install a utility called [figlets](http://www.figlet.org/) on your Mac. The figlets just make scanning long code blocks easier. If you have 15+ configuration groupings in your conditions file, the figlets make it easy to scan.

Next: 8. Customize the URL generator (page 36)

Summary: The sidebar and top navigation bar read their values from yml files. The navigation components are one of the most unique parts of this theme, since the navigation components are only included if they meet all of the product, audience, version, etc., values as specified in the project settings. Understanding how the sidebar works is critical to successfully using this theme.

Getting Started ▾

In the _data folder, the mydoc_sidebar.yml files contains the sidebar items for the theme. These list items (which are in YAML format) form your main navigation, and all pages in your project must appear here to be included in the PDF or the URL generator. Both the PDF and the URL generator (mydoc_urls.txt) iterate over the pages listed in the mydoc_sidebar.yml file to produce their output.

As a best practice, do the following with the sidebar:

- List all pages in your project somewhere in the sidebar.
- As soon as you create a new page, add it to your sidebar (so you don't forget about the page).
- Copy and paste the existing YAML chunks (then customize them) to get the formatting right.

YAML is a markup that uses spacing and hyphens instead of tags. YAML is a superset of JSON, so you can convert from YAML to JSON and vice versa equivalently.

There are certain values in the sidebar file coded to match the theme's code. These values include the main level names (`entries` , `subcategories` , `items` , `thirdlevel` , and `thirdlevelitems`). If you change these values in the sidebar file, the navigation won't display. (As long as you follow the sample with mydoc_sidebar.yml, you should be fine.)

At a high level, the sidebar data structure looks like this:

```
entries
  subcategories
    items
      thirdlevel
        thirdlevelitems
```

Within these levels, you add your content. You can only have two levels in the sidebar. Here's an example:

```
Introduction
 -> Getting started
 -> Features
 -> Configuration
    -> Options
    -> Automation
```

"Introduction" is a heading — it's the first level. Beneath it are Getting started and Features — these sub-items for the first level.

Configuration is a heading announcing a second level. Below it are Options and Automation — these are on the second level.

You can't add more than two levels. In general, it's a best practice not to create more than two levels of navigation anyway, since it creates a paralysis of choice for the user.

(If you need deeper sublevels, I recommend creating different sidebars for different pages, which is logic that I haven't coded into the theme but which could probably be added fairly easily. Additionally, if you wanted to create a third level, you could do so by following the same pattern as the second level and customizing a few things. However, the theme is not coded to support additional levels.)

The code in the theme's sidebar.html file (in the _includes folder) iterates through the items in the mydoc_sidebar.yml file using a Liquid `for` loop and inserts the items into HTML. Iterating over a list in a data file to populate HTML is a common technique with static site generators.

What I've added in this theme is some special logic that checks if the sidebar items meet the right attribute conditions. As a result, the sidebar.html file has code that looks like this:

```
 {% include custom/conditions.html %}
    {% for entry in sidebar %}
    {% for subcategory in entry.subcategories %}
        {% if subcategory.audience contains audience and subcat
egory.product contains product and subcategory.platform contain
s platform and subcategory.version contains version and subcate
gory.output contains "web" %}
```

Only if the sidebar item contains the right `audience`, `product`, `platform`, `version`, and `output` attributes does the item get included in the sidebar navigation.

This means you will have just one sidebar data file for all the outputs in a single project. Different projects will use different sidebar data files, but all outputs for a single project will use the same sidebar data file. (This allows you to do single sourcing.)

If you look at the code above, you'll see that `audience`, `product`, `platform`, and `version` are defined generally. In sidebar.html, there are lines like `subcategory.audience contains audience`.

This is where the conditions.html file (inside _includes) comes into play. `audience` is a variable defined in the conditions.html file. If you open up conditions.html, you'll see something like this:

```
{% if site.project == "mydoc_writers" %}
{% assign audience = "writers" %}
{% assign sidebar = site.data.mydoc.mydoc_sidebar.entries %}
{% assign topnav = site.data.mydoc.mydoc_topnav.topnav %}
{% assign topnav_dropdowns = site.data.mydoc.mydoc_topnav.topna
v_dropdowns %}
{% assign version = "all" %}
{% assign product = "all" %}
{% assign platform = "all" %}
{% assign projectTags = site.data.mydoc.mydoc_tags.allowed-tag
s %}
{% assign projectFolder = "mydoc" %}
{% endif %}
```

`audience` is a variable set to `writers` for the mydoc_writers project. Therefore anywhere `audience` appears, `writers` gets inserted in its place.

When the sidebar.html code runs `subcategory.audience contains audience`, it's saying that the subcategory item must have an attribute called `audience`, and the value for audience must contain `writers`.

All of the attributes (which are defined in the conditions.html file) must be met in order to display in the navigation. The attributes must be present on both the heading and items under that heading.

However, note that the `output` attribute is a bit different. With this attribute, you just list whether you want a `web` or `pdf` output (or both, usually). For both, you just write `web, pdf`.

The logic in the sidebar is multi-step and somewhat complex, but you're also doing something truly sophisticated. You're instructing a static site generator to conditionally include certain information while using the same source files (not just the same sidebar data file, but the same sidebar.html file).

Fortunately, once you set it up, you don't need to think about the underlying logic that's processing. You just make sure you're putting the right attributes on your sidebar items.

The first section in the sidebar subcategory list is a special frontmatter section that you should pretty much leave alone (except for changing the attribute values). It looks like this:

```
- title:
  audience: writers, designers
  platform: all
  product: all
  version: all
  output: pdf
  type: frontmatter
  items:
  - title:
    url: /titlepage.html
    audience: writers, designers
    platform: all
    product: all
    version: all
    output: pdf
    type: frontmatter
  - title:
    url: /tocpage.html
    audience: writers, designers
    platform: all
    product: all
    version: all
    output: pdf
    type: frontmatter
```

The only values you should change here are the values for the `audience`, `platform`, `product`, and `version`.

These frontmatter pages are used in producing the PDF. This part will grab the titlepage.html and tocpage.html content in the theme's root directory. (If you're not publishing PDF, you can remove this section.)

Note that the output is `pdf` only for these frontmatter pages. They are specific to the PDF output only.

To learn more about the sidebar, see Sidebar navigation (page 62).

Next: 5. Customize the conditions file (page 27)

Next: 7. Configure the top navigation (page 34)

Summary: The top navigation provides either single links or a drop-down menu. There are some other features, such as a feedback email, custom menu, and popout link.

Getting Started

The top navigation reads from the _data/mydoc/mydoc_topnav_doc.yml file. There are two *separate* sections in the mydoc_topnav_doc.yml file:

- `topnav`
- `topnav_dropdowns`

Items in the `topnav` section are rendered as single links. In contrast, items in the `topnav_dropdowns` section are rendered as a drop-down menu. You can't mix up the order of single links and drop-down links. The single links appear on the left, and the drop-down menus appear on the right.

If you click the Feedback link in the default theme, you'll see that it inserts the link to the current page along with a subject header and body. The topnav.html file contains an include to feedback.html. The feedback.html file contains the JavaScript that gets the current page URL and inserts it into the message body.

You configure the email in the configuration file through this property: `site.feedback_email` .

If you want the URL to point to an external site, use `external_url` instead of `url` in the data file. Then just enter the full HTTP URL. When you use `external_url` , the sidebar.html will apply this logic:

```
{% if item.external_url %}
<li><a href="{{item.external_url}}" target="_blank">{{subcatego
ry.title}}</a></li>
```

The way the PDF file is currently set up, only the links in the sidebar get included in the PDF. None of the links in the top nav get included in the PDF.

It wouldn't be hard to iterate through the top navigation bar and included the content in the PDF as well, but I think it's a best practice to put content links in the sidebar, and to put external links/resources in the top navigation.

If people open the site in a small browser, the top navigation will compress to a "hamburger." There's not a ton of room for adding links in this space.

Also note that the drop-down menus have one level only.

Summary: You need to customize the URL generator with your project's name. This generator helps you make quick links within your content.

Getting Started

The URL generator is a special file that helps you generate the code you need for links. This generator helps you avoid broken links, and ensures more consistency.

To learn more about the linking strategy used with this theme, see Links (page 89). The step here simply explains how to customize the URL generator for a new project.

1. In the project root directory, open urls_acme.txt (a file you should have already duplicated from urls_mydoc.txt) in an earlier step.
2. Do a find a replace for "mydoc" with "acme".
3. Change the project conditions at the top:

```
{% if site.project == "mydoc_writers" or site.project
== "mydoc_designers" %}
```

   Add all the projects here that will use this URL generator. For example, if you have 3 different projects, list them here. Otherwise the file's contents will be replaced with the values from the latest project that you run.

Notice that this URL generator iterates through the sidebar file only, and it doesn't apply the attribute qualifiers as with the sidebar.html file. As such, this URL generator will work with the output from any of the project files.

Next: 9. Set up Prince XML (page 37)

> **Summary:** Prince XML is the utility used for creating PDFs. Though not free, this utility gets a list of links and compiles them into a PDF.

> ☑ **Tip:** More details about generating PDFs are listed in Generating PDFs (page 115).

Prince XML is a utility I've decided to use to create PDFs. The Prince XML utility requires a list of web pages from which it can construct a PDF.

You need to install Prince. See the instructions on the Prince website

http://www.rince53 4492ETBT0025 5pions xml.of /doc/(You ne8 T/#macosxd(page 115)) 3 l034.8BT11320i0 5t74.7708 Td/F3util(Yo

Summary: You need to customize the build scripts. These script automate the publishing of your PDFs and web outputs through shell scripts on the command line.

Getting Started

The mydoc project has 5 build scripts and a script that runs them all. These scripts will require a bit of detail to configure. Every team member who is publishing on the project should set up their folder structure in the way described here.

Your command-line terminal opens up to your user name (for example, `Users/tjohnson`). I like to put all of my projects from repositories into a subfolder under my username called "projects." This makes it easy to get to the projects from the command line. You can vary from the project organization I describe here, but following the pattern I outline will make configuration easier.

To set up your projects:

1. Set up your Jekyll theme in a folder called "docs." All of the source files for every project the team is working on should live in this directory. Most likely you already either downloaded or cloned the jekyll-documentation-theme. Just rename the folder to "docs" and move it into the projects folder as shown here.
2. In the same root directory where the docs folder is, create another directory parallel to docs called doc_outputs.

   Thus, your folder structure should be something like this:

   ```
   projects
   - docs
   - doc_outputs
   ```

The docs folder contains the source of all your files, while the doc_outputs contains the site outputs.

For the mydocs project, you'll see a series of build scripts for each project. There are 5 build scripts, described in the following sections. Note that you really only need to run the last one, e.g., mydoc_all.sh, because it runs all of the build scripts. But you have to make sure each script is correctly configured so that they all build successfully.

☑ **Tip:** In the descriptions of the build scripts, "mydoc" is used as the sample project. Substitute in whatever your real project name is.

mydoc_1_multiserve_pdf.sh

Here's what this script looks like:

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


echo "Building PDF-friendly HTML site for Mydoc Writers ..."
jekyll serve --detach --config configs/mydoc/config_writers.ym
l,configs/mydoc/config_writers_pdf.yml
echo "done"

echo "Building PDF-friendly HTML site for Mydoc Designers ..."
jekyll serve --detach --config configs/mydoc/config_designers.y
ml,configs/mydoc/config_designers_pdf.yml
echo "done"

echo "All done serving up the PDF-friendly sites. Now let's gen
erate the PDF files from these sites."
echo "Now run . mydoc_2_multibuild_pdf.sh"
```

After killing all existing Jekyll instances that may be running, this script serves up a PDF friendly version of the docs (in HTML format) at the destination specified in the configuration file.

Each of your configuration files needs to have a destination like this: `../doc_outputs/mydoc/adtruth-java` . That is, the project should build in the doc_outputs folder, in a subfolder that matches the project name.

The purpose of this script is to make a version of the HTML output that is friendly to the Prince XML PDF generator. This version of the output strips out the sidebar, topnav, and other components to just render a bare-bones HTML representation of the content.

Customize the script with your own PDF configuration file names.

mydoc_2_multibuild_pdf.sh

Here's what this script looks like:

```
# Doc Writers
echo "Building the Mydoc Writers PDF ..."
prince --javascript --input-list=../doc_outputs/mydoc/writers-p
df/prince-file-list.txt -o mydoc/files/mydoc_writers_pdf.pdf;
echo "done"

# Doc Designers
echo "Building Mydoc Designers PDF ..."
prince --javascript --input-list=../doc_outputs/mydoc/designer
s-pdf/prince-file-list.txt -o mydoc/files/mydoc_designers_pdf.p
df;
echo "done"

echo "All done building the PDFs!"
echo "Now build the web outputs: . mydoc_3_multibuild_web.sh"
```

This script builds the PDF output using the Prince command. The script reads the location of the prince-file-list.txt file in the PDF friendly output folder (as defined in the previous script) and builds a PDF.

The Prince build command takes an input parameter (`--input-list=`) that lists where all the pages are (prince-file-list.txt), and then combines all the pages into a PDF, including cross-references and other details. The Prince build command also specifies the output folder (`-o`).

The prince-file-list.txt file (which simply contains a list of URLs to HTML pages) is generated by iterating through the table of contents (mydoc_sidebar.yml) and creating a list of URLs. You can open up prince-file-list.txt in the doc output to ensure that it has a list of absolute URLs (not relative) in order for Prince to build the PDF.

This is one way the configuration file for the PDF-friendly output differs from the HTML output. (If the PDF isn't building, it's because the prince-file-list.txt in the output is empty or it contains relative URLs.)

The Prince build script puts the output PDF into the mydoc/mydoc/files directory. Now you can reference the PDF file in your HTML site. For example, on the homepage you can allow people to download a PDF of the content at files/adtruth_dotnet_pdf.pdf.

mydoc_3_multibuild_web.sh

Here's what this script looks like:

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

echo "Building Mydoc Writers website..."
jekyll build --config configs/doc/config_writers.yml
# jekyll serve --config configs/doc/config_writers.yml
echo "done"

echo "Building Mydoc Designers website..."
jekyll build --config configs/doc/config_designers.yml
# jekyll serve --config configs/doc/config_designers.yml
echo "done"

echo "All finished building all the web outputs!!!"
echo "Now push the builds to the server with . mydoc_4_publis
h.sh"
```

After killing all Jekyll instances, this script builds an HTML version of the projects and puts the output into the doc_outputs folder. This is the version of the content that users will mainly navigate. Since the sites are built with relative links, you can browse to the folder on your local machine, double-click the index.html file, and see the site.

The `#` part below the `jekyll build` commands contains a serve command that is there for mere convenience in case you want to serve up just one site among many that you're building. For example, if you don't want to build everything — just one site — you might just use the serve command instead. (Anything after # in a YAML file comments out the content.)

mydoc_4_publish.sh

Here's what this script looks like:

```
echo "remove previous directory and any subdirectories without
a warning prompt"
ssh yourusername@yourdomain.com 'rm -rf /var/www/html/yourpubli
shingdirectory'

echo "push new content into the remote directory"
scp -r -vrC ../mydoc_outputs/doc-writers yourusername@yourdomai
n:/var/www/html/yourpublishingdirectory

echo "All done pushing doc outputs to the server"
```

This script assumes you're publishing content onto a Linux server.

Change `yourusername` to your own user name.

This script first removes the project folder on /var/www/html/yourpublishingdirectory site and then transfers the content from doc_outputs over to the appropriate folder in /var/www/html/yourpublishingdirectory.

Note that the delete part of the script (`rm -rf`) works really well. It annihilates a folder in a heartbeat and doesn't give you any warning prompts, so make sure you have it set up correctly.

Also, in case you haven't set up the SSH publishing without a password, see . Otherwise the script will stop and ping you to enter your password for each directory it transfers.

(Optional) Push to repositories

This script isn't included in the theme, but you might optionally decide to push the built sites into another github repository. For example, if you're using Cloud Cannon to deploy your sites, you can have Cloud Cannon read files from a specific Github repository.

Here's what this script looks like:

```
cd doc_outputs/mydoc/designers
git add --all
git commit -m "publishing latest version of docs"
git push
echo "All done pushing to Github"
echo "Here's the link to download the guides..."
cd ../../docs
```

This final script simply makes a commit into a Github repo for one of your outputs.

The doc_outputs/mydoc/designers contains the site output from mydoc, so when you push content from this folder into Github, you're actually pushing the HTML site output into Github, not the mydoc source files.

Your delivery team can also grab the site output from these repos. After downloading it, the person unzips the folder and sees the website folders inside.

mydoc_all.sh

Here's what this script looks like:

```
. deviceinsight_1_multiserve_pdf.sh; . deviceinsight_2_multibui
ld_pdf.sh; . deviceinsight_3_multibuild_web.sh; . deviceinsigh
t_4_publish.sh;
```

This script simply runs the other scripts. To sequence the commands, you just separate them with semicolons. (If you added the optional script, be sure to include it here.)

After you've configured all the scripts, you can run them all by running `. mydoc_all.sh` . You might want to run this script at lunchtime, since it may take about 10 to 20 minutes to completely build the scripts. But note that since everything is now automated, you don't have to do anything at all after executing the script. After the script finishes, everything is published and in the right location.

After setting up and customizing the build scripts, run a few tests to make sure everything is generating correctly. Getting this part right is somewhat difficult and may likely require you to tinker around with the scripts a while before it works flawlessly.

Summary: This theme uses pages only, not posts. 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.

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 project 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.

By default, everything in your project is included in the output. This is problematic when you're single sourcing and need to exclude some files from an output.

Here's the approach I've taken. Put all files in your root directory, but put the project name first and then any special conditions. For example, mydoc_writers_intro.md.

In your configuration file, you can exclude all files that don't belong to that project by using wildcards such as the following:

exclude:

- mydoc_*
- mydoc_writers_*

These wildcards will exclude every match after the `*`.

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

```
---
title: Your page title
tags: [formatting, getting_started]
keywords: overview, going live, high-level
last_updated: November 30, 2015
summary: "Deploying DeviceInsight requires the following step
s."
---
```

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

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

FRONTMATTER	REQUIRED?	DESCRIPTION
title	Required	The title for the page
tags	Optional	Tags for the page. Make all tags single words, with hyphens 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.
keywords	Optional	Synonyms and other keywords for the page. This information gets stuffed into the page's metadata to increase SEO. The user won't see the keywords, but if you search for one of the keywords, it will be picked up by the search engine.
last_updated	Optional	The date the page was last updated. This information could helpful for readers trying to evaluate how current and authoritative information is. If included, the last_updated date appears in the footer of the page.
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.)
datatable	Optional	Boolean. If you add `true`, then scripts for the jQuery datatables plugin (https://www.datatables.net/) get included on the page.

FRONTMATTER	REQUIRED?	DESCRIPTION
video	Optional	If you add `true`, then scripts for Video JS: The HTML5 video player (http://www.videojs.com/) get included on the page.

☑ **Tip:** You can see the scripts that conditionally appear by looking in the _layouts/default.html page. Note that these scripts are served via a CDN, so the user must be online for the scripts to work. However, if the user isn't online, the tables and video still appear. In other words, they degrade gracefully.

What about permalinks? This theme isn't build using permalinks because it makes linking and directory structures problematic. Permalinks generate an index file inside a folder for each file in the output. This makes it so links (to other pages as well as to resources such as styles and scripts) need to include `../` depending upon where the other assets are located. But for any pages outside folders, such as the index.html page, you wouldn't use the `../` structure.

Basically, permalinks complicate the linking structure significantly, so they aren't used here. As a result, page URLs have an .html extension. If you include `permalink: something` in your frontmatter, your link to the page will break (actually, you could still go to sample instead of sample.html, but none of the styles or scripts will be correctly referenced).

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

If you add `published: false` in the frontmatter, your page won't be published. You can also move draft pages into the _drafts folder to exclude them from the build.

☑ **Tip:** You can create file templates in WebStorm that have all your common frontmatter, such as all possible tags, prepopulated. See WebStorm Text Editor (page 50) for details.

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 — if you use HTML (as your file extension), you can't insert Markdown content.

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.

Store all your pages inside the root directory. This is because the site is built with relative links. There aren't any permalinks or baseurls used in the link architecture. This relative link nature of the site allows you to easily move it from one folder to another without invalidating the links.

If this approach creates too many files in one long list, consider grouping files into Favorites sections using WebStorms Add to Favorites feature.

You can use standard Multimarkdown syntax for tables. You can also use fenced code blocks. The configuration file shows the Markdown processor and extensiosn:

```
markdown: redcarpet

redcarpet:
  extensions: ["no_intra_emphasis", "fenced_code_blocks", "tabl
es", "with_toc_data"]
```

These extensions mean the following:

REDCARPET EXTENSION	DESCRIPTION
no_intra_emphasis	don't italicize words with underscores
fenced_code_blocks	allow three backticks before and after code blocks instead of `<pre>` tags
tables	allow table syntax
with_toc_data	add ID tags to headings automatically

You can also add "autolink" as an option if you want links such as http://google.com to automatically be converted into links.

Note: Make sure you leave the `with_toc_data` option included. This auto-creates an ID for each Markdown-formatted heading, which then gets injected into the mini-TOC. Without this auto-creation of IDs, the mini-TOC won't include the heading. If you ever use HTML formatting for headings, you need to manually add an ID attribute to the heading in order for the heading to appear in the mini-TOC.

By default, a mini-TOC appears at the top of your pages and posts. If you don't want this, you can remove the `{% include toc.html %}` from the layouts/page.html file.

If you don't want the TOC to appear for a specific page, add `toc: false` in the frontmatter of the page.

The mini-TOC requires you to use the `##` syntax for headings. If you use `<h2>` elements, then you must add an ID attribute for the h2 element in order for it to appear in the mini-TOC.

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.

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 code isn't included.

This theme isn't coded with any kind of posts logic. For example, if you wanted to add a blog to your project that leverages posts, you couldn't do this with the theme. However, you could easily take the post logic from another site and integrate it into this theme. I've just never had a strong need to integrate blog posts into documentation.

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, when I type `jif`, aText replaces it with `{% if site.platform == "x" %}`. When I type `jendif`, aText replaces it with `{% endif %}`.

You get aText from the App Store on a Mac for about $5.

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

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.

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.

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.

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

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".)

COMMAND	SHORTCUTS
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.

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.

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.

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.

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.

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.

You can then incorporate conditional statements that check the values in the configuration files.

☑ **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.

You can filter content based on values that you have set either in your 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, you don't need to restart the server when you make changes.

This theme requires you to add the following attributes in your configuration file:

- project
- audience
- product
- platform

- version

If you've ever used DITA, you probably recognize these attributes, since DITA has mostly the same ones. I've found that most single_sourcing projects I work on can be sliced and diced in the ways I need using these conditional attributes.

If you're not single sourcing and you find it annoying having to specify these attributes in your sidebar, you can rip out the logic from the sidebar.html, topnav.html file and any other places where conditions.html appears; then you wouldn't need these attributes in your configuration file.

Here's an example of conditional logic based on a value in the configs/config_writer.yml file. In my config_writer.yml file, I have the following:

```
audience: writers
```

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

```
{% if site.audience == "writers" %}
The writer audience should see this...
{% elsif site.audience == "designers" %}
The designer audience should see this ...
{% 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 site.audience == "writer" %}
2. Add in a pound of meat.
{% elsif site.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`).

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 site.audience contains "vegan" or site.audience == "veget
arian" %}
    // 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 site.audience contains "vegan" or "vegetarian" %}
    // run this.
{% endif %}
```

This won't work.

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

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

When figuring out this logic, read it like this: "Run the code here *unless* this condition is satisfied." Or "If this condition is satisfied, don't run this code."

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.)

In this situation, if `site.print == true`, then the code will *not* be run here.

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.

I don't really use the _data folder as much for project options. I store them in the configuration file because I usually want different projects to use different values for the same property.

For example, maybe a file or function name is called something different for different audiences. I currently single source the same content to at least two audiences in different markets.

For the first audience, the function name might be called `generate` , but for the second audience, the same function might be called called `expand` . In my content, I'd just use `{{site.function}}` . Then in the configuration file I change its value appropriately for the audience.

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

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

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

```
data_source: data_alpha
```

Then create a folder called _data_alpha.

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

```
data_source: data_beta
```

Then create a folder called _data_beta.

You can also create conditional logic based on the page namespace. For example, create a page with front matter as follows:

```
---
layout: page
user_plan: full
---
```

Now you can run logic based on the conditional property in that page's front matter:

```
{% if page.user_plan == "full" %}
// run this code
{% endif %}
```

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.

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.

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/mydoc folder (replacing "mydoc" with your project's name), and then use a tag like this:

```
{% include mydoc/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 on that.

When you want to re-use a topic across projects, store the content in the \includes folder (it can be in any project's subfolder). Any folder that begins with an underscore (_) isn't included in the site output.

Also be sure to put any images in the common_images folder. None of the assets in the common_images folder should be excluded in the configuration files. This means every project's output will include the resources from the common_images folder.

However, each project will likely exclude content from the specific folders where the pages are stored. This is why reuse across projects requires you to use the _includes folder and the common_images folder. (Unfortunately you can't include an image from the _includes folder.)

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 haven't found a use case for page-level variables, but it's nice to know they're available.

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.

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.

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/2015/02/20/jekyll-collections/).

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.

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.

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 130) for details.

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.

Note: For basic information about configuring the sidebar navigation, see . This section gets into the top sidebar navigation in more depth.

When you set up your project, you configured the sidebar following the instructions in . In this topic, I dive deeper into other aspects of the sidebar.

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_doc.yml 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.)

As mentioned in the previous section, the theme uses the Navgoco sidebar (http://www.komposta.net/article/navgoco). 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`.

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.

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.

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/mydoc_sidebar_navigation.html. The `url` attribute in the sidebar item must match the page URL in order to get the `active` class applied.

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

```
    - title: Understanding how the sidebar works
      url: /mydoc/mydoc_understand_sidebar.html
      audience: writers, designers
      platform: all
      product: all
      version: all
      output: web, pdf
```

Note that the url includes the project folder where the file is stored.

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.

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.

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]
---
```

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` name here.

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

Note: Unlike with WordPress, 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).

Tags have a few components.

1. First make sure you configure a few details in the conditions.html file. In particular, see this setting:

   ```
   {% assign projectTags = site.data.tags_doc.allowed-tags
   %}
   ```

   The tags_doc name must correspond with how you label your tags file. Here, "doc" should be your project name.

2. In the _data file, add a yml file similar to tags_doc.yml. The YML file lists the tags that are allowed:

   ```
   allowed-tags:
     - getting_started
     - overview
     - formatting
     - publishing
     - single_sourcing
     - special_layouts
     - content types
   ```

3. Create a tag archive file for each tag in your tags_doc.yml list. Name the file like this: tag_getting_started.html, where doc is your project name. (Again, tags with multiple words need hyphens in them.)

   Each tag archive file needs only this:

   ```
   ---
   title: "Getting Started Pages"
   tagName: getting_started
   ---
   {% include taglogic.html %}
   ```

> ℹ **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.

4. Adjust button color or tag placement as desired.

   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>
{% include custom/conditions.html %}
{% for tag in page.tags %}
{% if projectTags contains tag %}
<a href="tag_{{tag}}.html" class="btn btn-info navbar-bt
n cursorNorm" role="button">{{page.tagName}}{{tag}}</a>
{% endif %}
{% endfor %}
{% endif %}
</div>
```

Because this code appears on the _layouts/page.html file by default, you don't need to do anything. However, if you want to alter the placement or change the button color, you can do so.

You can change the button color by changing the class on the button from `btn-info` to one of the other button classes bootstrap provides. See Labels (page 88) for more options on button class names.

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 1)
- About the theme author (page 0)
- 2. Add a new project (page 12)
- 6. Configure the sidebar (page 29)
- 3. Decide on your project's attributes (page 15)
- 1. Build the default project (page 8)
- Pages (page 44)
- Sidebar Navigation (page 62)
- Support (page 0)
- Supported features (page 3)
- Troubleshooting (page 169)
- WebStorm Text Editor (page 50)

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:

- 1. Build the default project (page 8)
- 2. Add a new project (page 12)
- 3. Decide on your project's attributes (page 15)
- 6. Configure the sidebar (page 29)
- About the theme author (page 0)
- Introduction (page 1)
- Pages (page 44)
- Sidebar Navigation (page 62)
- Support (page 0)
- Supported features (page 3)
- Troubleshooting (page 169)
- WebStorm Text Editor (page 50)

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.

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.

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 50) for tips on creating file templates in WebStorm.

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

You create a series by looking for all pages within a tag namespace that contain certain frontmatter. Here's a demo.

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/mydoc 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.

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: "0.1"  %}
    {% 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.

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

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

With weight, you could use 1, 2, 3, etc.., but Jekyll will treat 10 as coming after 1. This is why I use 1.0 and 1.1, 1.2, etc.

If you do use whole numbers, change the `plus: "0.1"` to `plus: "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.

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/mydoc/series_acme.html %}

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

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

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 88) for more Bootstrap button classes.

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 60).

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.

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.

Summary: You can insert notes, tips, warnings, and important alerts in your content. These notes are stored as shortcodes made available through the linksrefs.hmtl include.

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.

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.

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.

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 call-out_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.

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.

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.

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.

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).

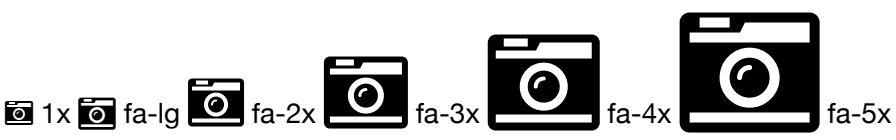
Go to the Font Awesome library (http://fortawesome.github.io/Font-Awesome/icons/) to see the available icons.

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
```

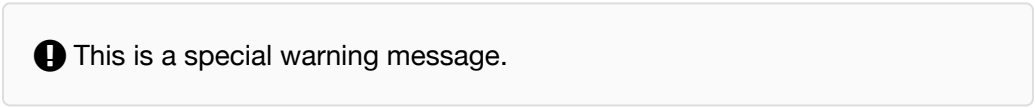
Here's what they render to:



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:



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}}
```

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.

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

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

And the result:

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

Grab the right class name from the Font Awesome library (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%.

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>
```

Error: Enter a valid email address

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>
```

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}}<div class="bs-callout bs-call
out-info">{{site.data.alerts.end}}
```

You can use any of the following:

```
{{callout_danger}}
{{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}}
```

Callouts are explained in a bit more detail here: Alerts (page 75).

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" />
```

And the result:

Here's the same Markdown syntax:

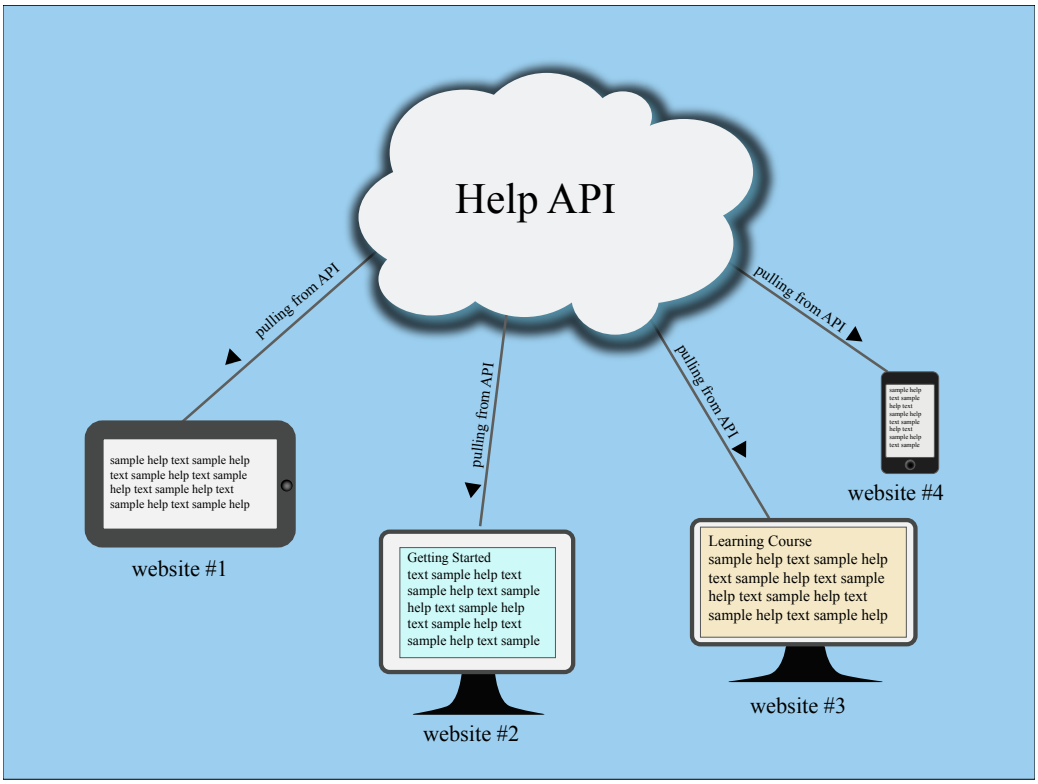
```
![My sample page](images/jekyll.png)
```

And the result:

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="images/dicloud/dicloud_architecture.svg" style="max-width: 700px;" />
```

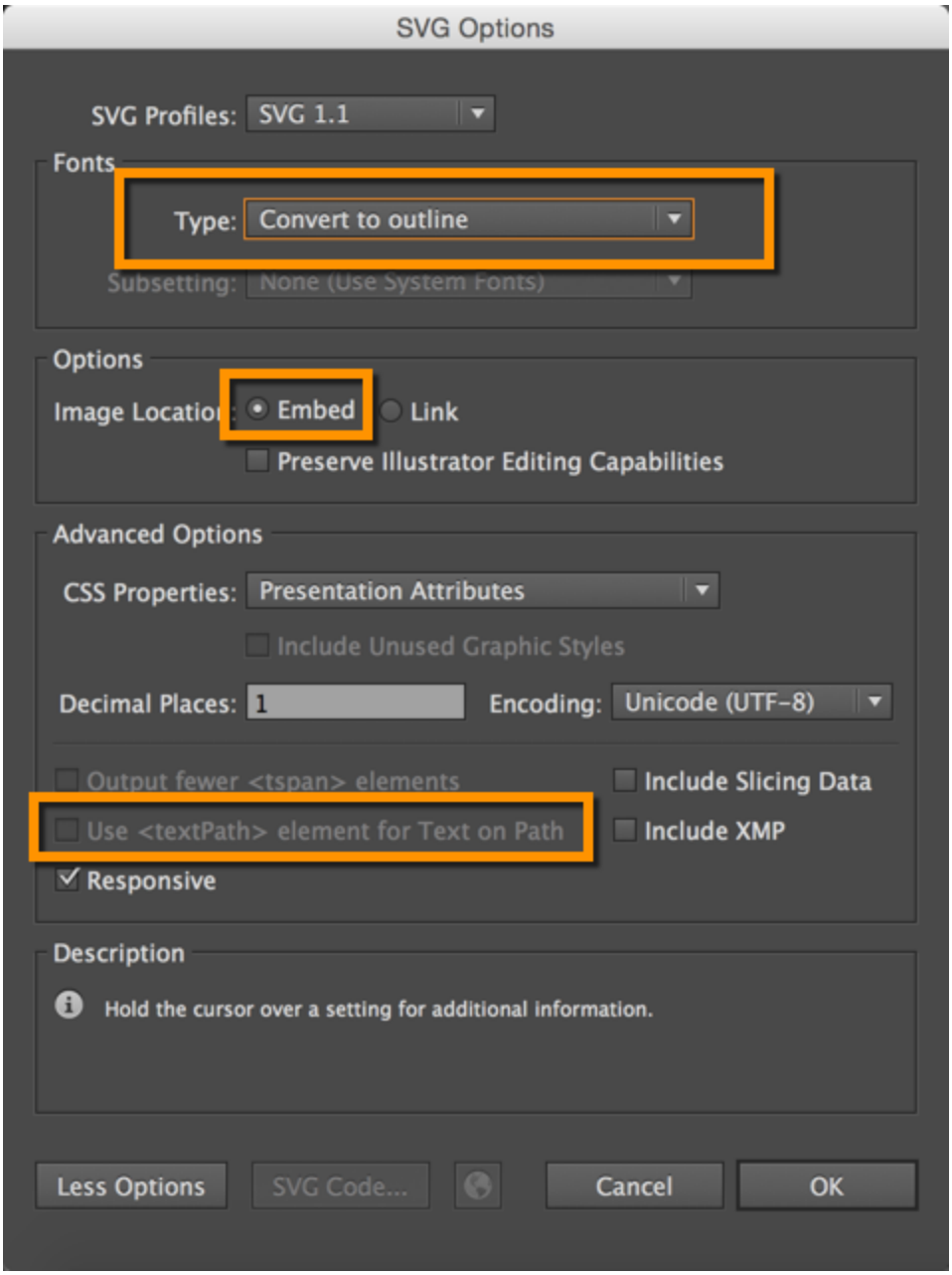
Here's the result:



SVG images will expand to the size of their artboard, so you can either set the artboard the right size when you create the graphic in Illustrator, or you can set an inline style that confines the size to a certain width as shown in the code above.

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.


SVG Options
SVG Profiles: SVG 1.1
Fonts
Type: Convert to outline
Subsetting: None (Use System Fonts)
Options
Image Location: Embed Link
Preserve Illustrator Editing Capabilities
Advanced Options
CSS Properties: Presentation Attributes
Include Unused Graphic Styles
Decimal Places: 1
Encoding: Unicode (UTF-8)
Output fewer <tspan> elements
Include Slicing Data
Use <textPath> element for Text on Path
Include XMP
Responsive
Description
Hold the cursor over a setting for additional information.
Less Options
SVG Code...
Cancel
OK

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.

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.

Summary: When creating links, although you can use standard HTML or Markdown, this approach is usually susceptible to a lot of errors and broken links. There's a URL generator that will facilitate linking to other pages in ways that ensures the links won't break.

One of the more difficult parts of a documentation site is keeping all the internal links accurate and valid. When you're single sourcing, you usually have multiple documentation outputs that include certain pages for certain audiences. Orphan links are a common problem to avoid.

Although there are many ways to create links, I'll just describe what I've found to work well.

When linking to an external site, use Markdown formatting:

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

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

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

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

```
[Sample](sample.html)
```

OR

```
<a href="sample.html">Sample</a>
```

However, what happens when you change the page's title or link? Jekyll doesn't automatically pull in the page's title when you create links.

In my experience, coding links like this results in a lot of broken links.

For internal links, I've found that it's a best practice to store the link in a YAML file that is derived from the table of contents.

The theme has a file called urls_mydoc.txt. This file contains the same code as the table of contents (but without the conditional qualifiers). It iterates through every page listed in the table of contents sidebar (as well as the top navigation menus) and creates an output that looks like this for each link:

```
mydoc_getting_started:
  title: "Getting started with this theme"
  url: "mydoc_getting_started.html"
  link: "<a href='mydoc_getting_started.html'>Getting started w
ith this theme</a>"
```

From the site output folder (in ../doc_outputs), open urls_mydoc.txt and observe that it is properly populated (blank spaces between entries doesn't matter). Then manually copy the contents from the mydoc_urls.txt and insert it into the _data/mydoc/mydoc_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.mydoc.mydoc_urls.mydoc_getting_started.link}}
```

This will insert the following into your topic:

```
<a href='mydoc_getting_started.html'>Getting started with this
theme</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.mydoc.mydoc_urls.mydoc_getting_started.ur
l}}).
```

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.my
doc.mydoc_urls.mydoc_getting_started.url}}">get started in buil
ding the theme</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).

You should treat your sidebar_doc.yml file with a lot of care. Every time you add a page to your site, make sure it's listed in your sidebar_doc.yml file (or in your top navigation). If you don't have pages listed in your sidebar_doc.yml file, they won't be included in the urls_mydoc.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.

Another way to ensure you don't have any broken links in your output is to generate a PDF (page 115). 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.

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

The site is coded with relative links. There aren't any permalinks, urls, or baseurls. The folder structure you see in the project directory is the same folder directory that gets built in the site output.

Author all pages in your root directory. This greatly simplifies linking. However, when you're linking to images, files, or other content, you can put these assetserence into aRTd[bg tod[org

Jekyll gets around some of this link path variation by using `baseurl` and including code that prepends the `baseurl` before a link. This converts the links into absolute rather than relative links.

With absolute links, the site only displays at the `baseurl` you configured. This is problematic for tech docs because you usually need to move files around from one folder to another based on versions you're archiving or when you're moving your documentation from draft to testing to production folders.

One of the shortcomings in this theme is that the link titles in the sidebar and inline links don't necessarily have to match the titles specified on each page. You have to manually keep the page titles in sync with the titles listed in the sidebar and top navigation. Although I could potentially get rid of the titles key in the article topic, it would make it more difficult to know what page you're editing.

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.

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.

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

Profile About Match

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.

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>
```

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.

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.

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.

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">` .

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">
```

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)
.

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).

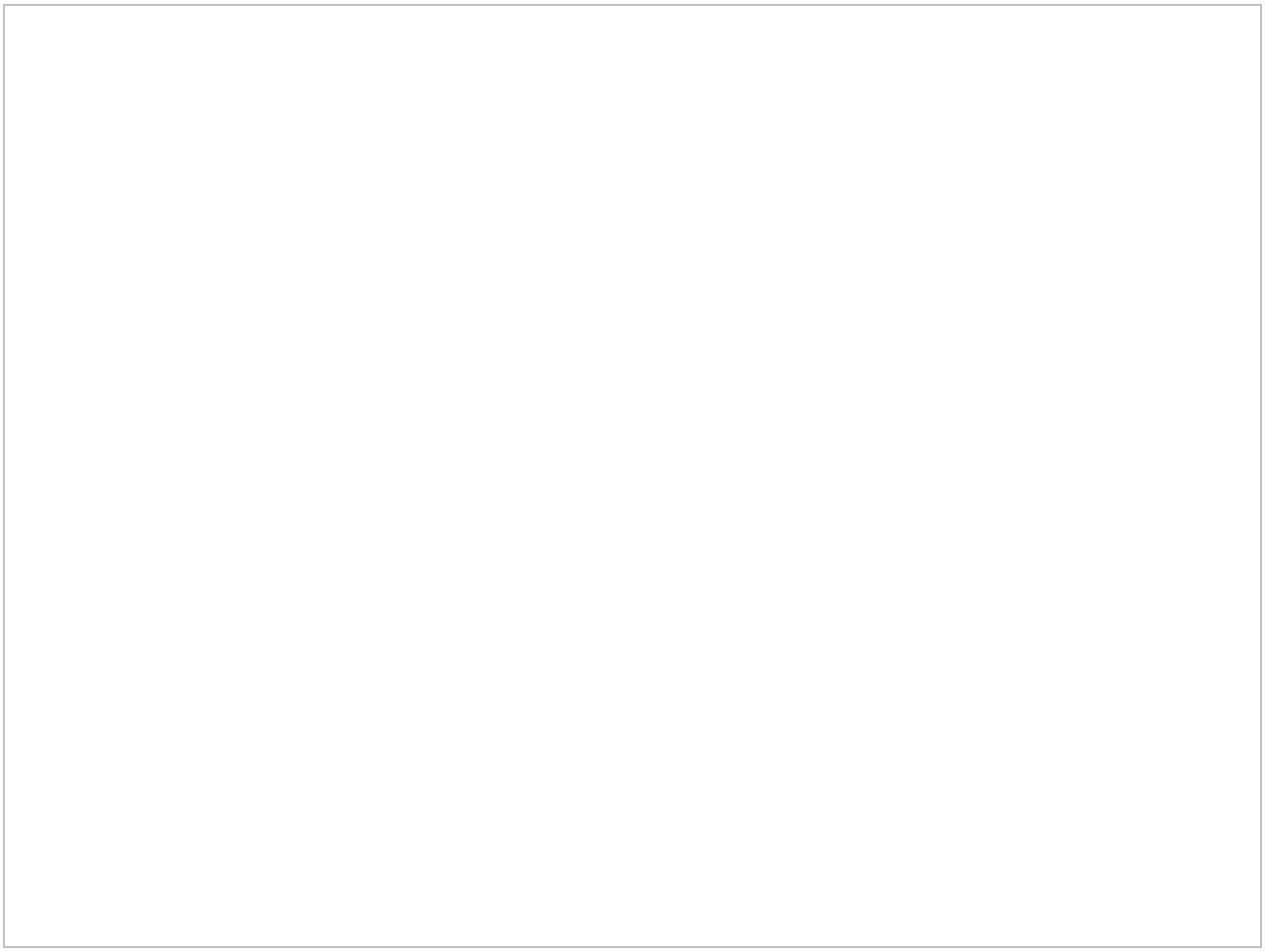
One piece of functionality I'd like to implement is the ability to set site-wide nav

Summary: You can embed files with a Video JS wrapper by adding 'video: true' in the frontmatter. Alternatively, you can just fall back on the default video wrapper in the browser.

The theme has the video.js (http://www.videojs.com/) player integrated. But the scripts only appear on a page or post if you have certain frontmatter in that page or post. If you want to embed a video in a page and use the Video JS player, add `video: true` in your frontmatter of a page or post, and then add code like this where you want the video to appear:

```
<p><video id="scenario-1" class="video-js vjs-default-skin vj
s-big-play-centered" controls
 preload="auto" width="640" height="480" data-setup='{}'>
  <source src="http://idratherbetellingstories.com/podcasts/ont
ariochapterpresentation/ontariochapterv4.mp4" type='video/mp4'>
</video></p>
```

Here's an example:



If you want the player button in the upper-left corner (which is the default), remove the `vjs-big-play-centered` from the video class.

Here are more details on this video player from Video JS (https://github.com/videojs/video.js/blob/stable/docs/guides/setup.md).

Note that if some of the js doesn't load correctly, the default fallback player is the regular HTML5 video player available via the browser. Here's an example of the built-in browser video wrapper:

Your browser does not support the video tag.

However, I don't think the built-in browser video players work very well (you can't easily scrub around the video without seeing lots of buffering and other issues). But definitely compare the two. You may find that adding the Video JS wrapper is overkill.

Warning: Github wasn't designed to store video content. If you have an mp3 file, don't store it in your Github directory. Instead, put it on a web host using regular FTP methods, or stream the video from a video streaming

service such as Youtube or Vimeo. Also, note that Github's Large File Storage (which does handle large files) isn't compatible with Github Pages.

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.

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

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: true` 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.

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.

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

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:

```ruby
    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: pygments
```

You can use another highlighter such as `rouge`.

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

The keywords you must add to specify the highlighting (in the previous example, `ruby`) are called "lexers." You can search for "pygments lexers" or go directly to Available lexers (http://pygments.org/docs/lexers/) to see what values you can use. Here are some common ones I use:

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

Summary: You can add a button to your pages that allows people to add comments. Prose.io is an overlay on Github that would allow people to make comments in an easier interface.

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:

```
    <a href="https://github.com/tomjohnson1492/documentation-th
eme-jekyll/edit/reviews/mydoc/mydoc_commenting_on_files.md" cla
ss="btn btn-default " role="button"><i class="fa fa-github fa-l
g"></i> Edit me</a>
```

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.

Summary: When you have a single sourcing project, you use more advanced arguments when you're building or serving your Jekyll sites. These arguments specify a particular configuration file and may build on other configuration files.

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/config_writers.yml --destination
/users/tjohnson/projects/documentation-theme-jekyll-builds/writ
er
```

Here the `configs/config_writers.yml` file is used instead of `_config.yml`. The destination directory is `../mydoc_writers`.

If you don't want to enter the long Jekyll argument every time, with all your configuration details, you can create a shell script and then just run the script. This theme shows an example with the mydoc_multibuild_web.sh file in the root directory.

My preference is to add the scripts to profiles in iTerm. See iTerm profiles (page 144) for more details.

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
```

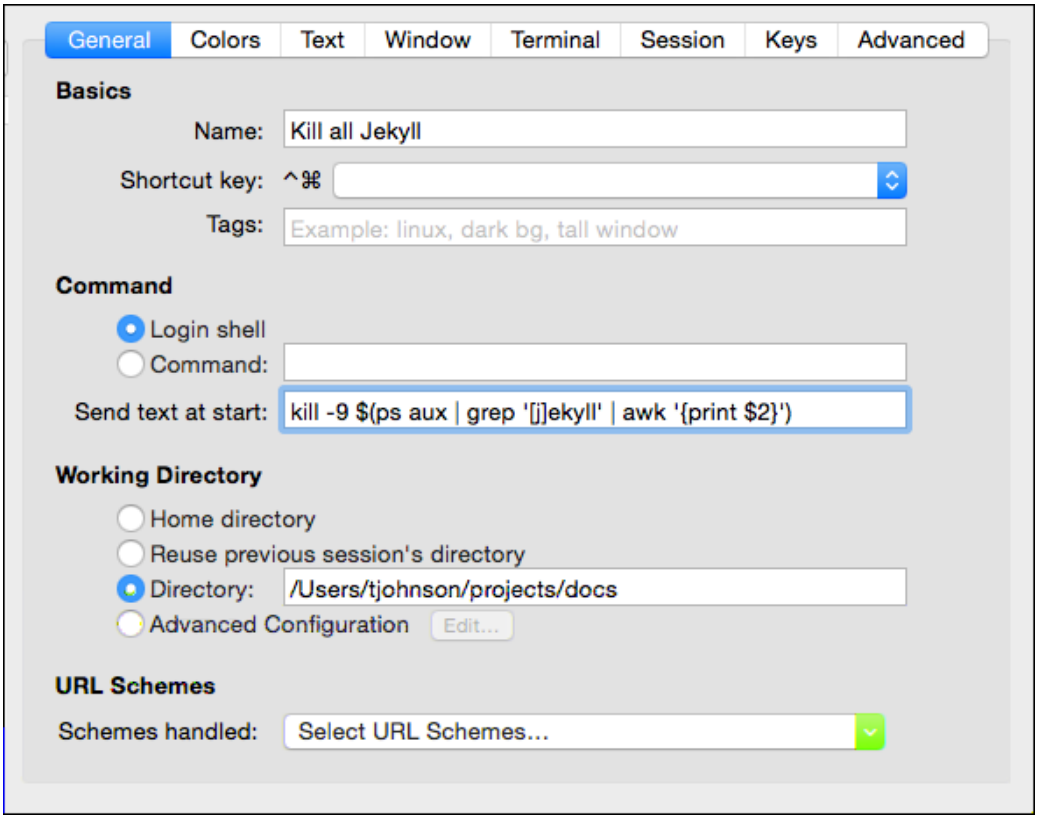
Find the PID (for example, it looks like "22298").

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 created 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...

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.

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 configuration file, specify the theme file you want the output to use — for example, `theme_file: theme-green.css`.

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.

Summary: Before deploying your published site, you want to ensure that you don't have any broken links. There are a few ways to check for broken links.

One of the challenging aspects of technical writing is avoiding broken links in your output. Consider this example. You have three outputs, with different topics included for different audiences. The topics each have inline cross references pointing to the other topics, but since some of the topics aren't included for each audience, you risk having a broken link for the output that omits that topic.

Additionally, technical writers frequently manage large numbers of topics, and as they make updates, they rename titles, remote some topics, combine multiple topics into the same topic, and make other edits. When you're developing content, the pages and titles in your topics and navigation are in flux. You shift things around constantly trying to find the right organization, the right titles, and more.

During this time, if you have inline links that point to specific pages, how do you avoid broken links in your output?

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.html 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.

When you generate a PDF, Prince XML will print "page 0" for any cross references it can't find. This lets you know that a particular link is bad because the page is missing.

If you have links in your PDF that aren't references to other topics (maybe they're links to PDF file assets, or links within a navtab or collapsible section), then you must add a class of `noCrossRef` to the link to avoid having Prince write "page 0" for the link.

(Note that there are still some kinks I'm working out with this. You may find that links still say "page 0" even if they have the `noCrossRef` class.)

Instead of creating links to direct pages, use the data reference technique described in Links (page 89). With this method, the urls.txt file iterates through all the pages in your navigation and formats them into a YAML syntax.

Then you insert an inline link by referring to that YAML data. For example, the previous hyperlink is `{{site.data.mydoc.mydoc_urls.mydoc_hyperlinks.link}}`.

As you go through the link validation process, make sure you copy over the content from the generated urls.txt (in the Jekyll site output) and insert it into the urls.yml file in your _data folder.

Summary: title checker...

To make sure your page titles match the sidebar titles, there's a file called "title-checker.html." After your site builds, view this file in your browser. It will tell if you any of your page titles don't match up with your sidebar titles.

This utility isn't 100% comprehensive, though. If your sidebar has a URL that doesn't match any file name in your project, it won't be able to handle that URL. Therefore, when you do validity checks for the links on your site, click through each item in the sidebar to make sure the page loads.

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.

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.

You can see an example of the finished product here:

PDF Download

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.

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 config_designers.yml file for this theme:

```
destination: ../doc_outputs/mydoc/designers-pdf
url: "http://127.0.0.1:4010"
baseurl: "/mydoc/designers-pdf"
port: 4010
output: pdf
print_title: Jekyll theme for documentation – designers
print_subtitle: version 4.0
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. Prince won't be able to find files if they simply have relative paths, such as /sample.html. The must have full URLs it can access — hence the `url` and `baseurl`.

Also note that the default page layout 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.

Finally, 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.

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.

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 theme, sidebar_doc.yml):

```
- title:
  audience: writers, designers
  platform: all
  product: all
  version: all
  output: pdf
  type: frontmatter
  items:
  - title:
    url: /titlepage.html
    audience: writers, designers
    platform: all
    product: all
    version: all
    output: pdf
    type: frontmatter
  - title:
    url: /tocpage.html
    audience: writers, designers
    platform: all
    product: all
    version: all
    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 nearly identical to that of the sidebar.html page except that it includes the `site` and `baseurl` for the URLs. 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-file-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-file-list.txt and create a running PDF that contains all of the pages listed, with appropriate cross references and styling for them all.

> ℹ **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-file-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.

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 cool. 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.

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[hr
ef].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 following 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: December 02, 2015";
    });
</script>

<script>
    Prince.addScriptFunc("guideName", function() {
        return "Jekyll theme for documentation – writers User G
uide";
    });
</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).

Open the mydoc_1_multiserve_pdf.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 Writers ..." jekyll serve --detach --config configs/mydoc/config_writers.yml,configs/mydoc/config_writers_pdf.yml echo "done"

echo "Building PDF-friendly HTML site for Mydoc Designers ..." jekyll serve --detach --config configs/mydoc/config_designers.yml,configs/mydoc/config_designers_pdf.yml echo "done"

echo "All done serving up the PDF-friendly sites. Now let's generate the PDF files from these sites." echo "Now run . mydoc_2_multibuild_pdf.sh"

Note that the first part kills all Jekyll instances. This way you won't try to server 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.

Open up mydoc_2_multibuild_pdf.sh and look at the Prince commands:

```
# Doc Writers
echo "Building the Mydoc Writers PDF ..."
prince --javascript --input-list=../doc_outputs/mydoc/writers-p
df/prince-file-list.txt -o mydoc/files/mydoc_writers_pdf.pdf;
echo "done"

# Doc Designers
echo "Building Mydoc Designers PDF ..."
prince --javascript --input-list=../doc_outputs/mydoc/designer
s-pdf/prince-file-list.txt -o mydoc/files/mydoc_designers_pdf.p
df;
echo "done"

echo "All done building the PDFs!"
echo "Now build the web outputs: . mydoc_3_multibuild_web.sh"
```

This 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-file-list.txt file. Then we tell it where and what to output (`-o`).

Make sure that the path to the prince-file-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.

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

```
<a target="_blank" class="noCrossRef" href="files/mydoc_writer
s_pdf.pdf"><button type="button" class="btn btn-default" aria-l
abel="Left Align"><span class="glyphicon glyphicon-download-al
t" aria-hidden="true"></span> PDF Download</button></a>
```

Here's what that looks like:

<a target="_blank" class="noCrossRef" href="files/mydoc_writers_pdf.pdf"><button type="button" class="btn btn-default" aria-label="Left Align"> PDF Download</button></a>

The `{{site.pdf_file_name}}` {% raw %} is set in the configuration file.

{{site.data.alerts.note}} If you don't like the style of the PDFs, just adjust the styles in the printstyles.css file.{{site.data.alerts.end}}

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

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

However, the PDF will still build.

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

Then surround the JavaScript with conditional tags like this:

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

For more detail about using `unless` in conditional logic, see Conditional logic (page 53). What this code means is "run this code unless this value is the case."

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.

Summary: By default, all the files in your Jekyll project are included in the output (this differs from DITA projects, which don't include files unless noted on the map). If you're single sourcing, you'll need to exclude the files that shouldn't be included in the output. The sidebar doesn't control inclusion or exclusion.

By default, all files in your project are included in your output (regardless of whether they're listed in the sidebar_doc.yml file or not). To exclude files, note them in the `exclude` section in the configuration file. Here's a sample:

```
exclude:
  - mydoc_writers_*
  - bower_components
  - Gemfile
```

If you have different outputs for your site, you'll want to customize the exclude sections in your various configuration files.

Here's the process I recommend. Put all files in the root directory of your project. Suppose one project's name is alpha and the other is beta. Then name each file as follows:

- alpha_sample.html
- beta_sample.html

In your exclude list for your beta project, specify it as follows:

```
exclude:
- alpha_*
```

In your exclude list for your alpha project, specify it as follows:

```
exclude:
- beta_*
```

If you have more sophisticated exclusion, add another level to your file names. For example, if you have different programming languages you want to filter by, add this:

- alpha_java_sample.html
- alpha_cpp_sample.html

Then you exclude files for your Alpha C++ project as follows:

```
exclude:

- alpha_java_*
- beta_*
```

And you exclude files for your Alpha Java project as follows:

```
exclude:

- alpha_cpp_*
- alpha_beta_*
```

When you exclude folders, include the trailing slash at the end of the folder name:

```
exclude:
- images/alpha/
```

There isn't a way to automatically exclude anything. By default, everything is included unless you explicitly list it under the exclude section.

If you're working on a draft, put it inside the _drafts folder or add `published: false` in the frontmatter. The _drafts folder is excluded by default, so you don't have to specify it in your exclude list.

What if a file should appear in two projects but not the third? This can get tricky. For some files, rather than using a wildcard, you may need to manually specify the entire filename that you're excluding instead of excluding it by way of a wildcard pattern.

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.

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

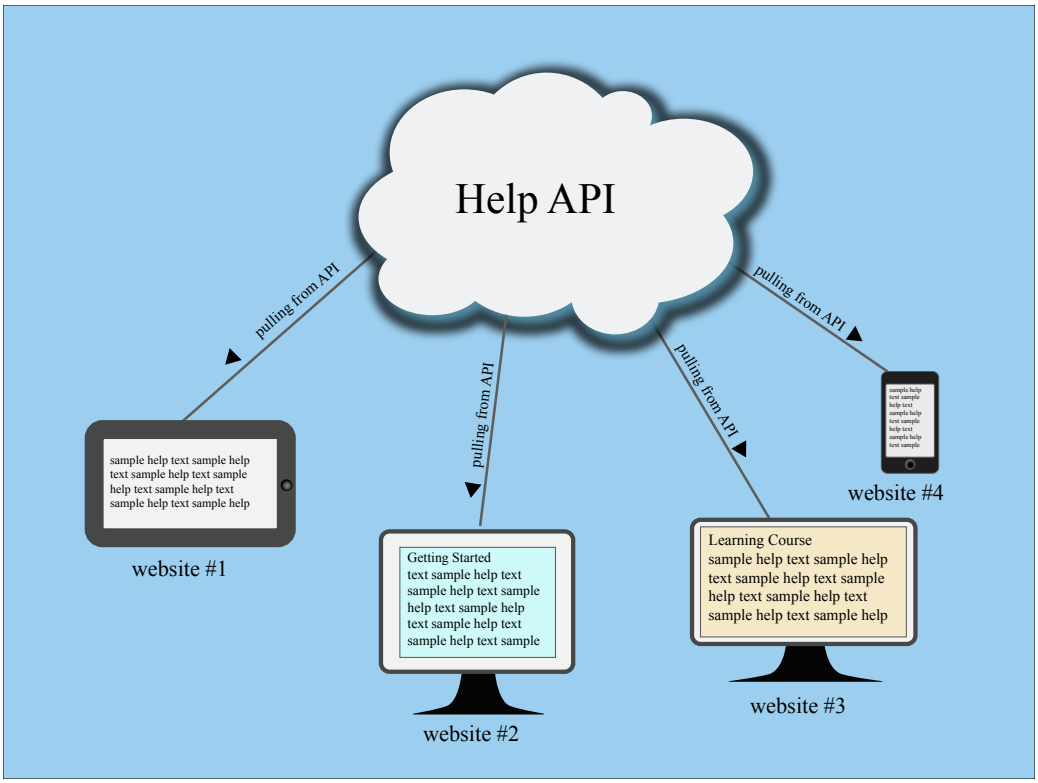
For the full code demo, see the notes in the tooltip demo.

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.

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.

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`.

Inside _data > mydoc create a YAML file called something like mydoc_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.

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 mydoc_definitions.yml file.

Here's an example:

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

{{site.data.mydoc.mydoc_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.

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.

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.

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/apidemo
s/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.

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 I.3662 TdE85ods7313.8765 502.6835 Td

The `each` method looks through all the JSON content to find the item whose `page.id` is equal to `basketball`. It then looks for an element on the page named `#basketball` and adds a `data-content` attribute to that element.

> ❗ **Warning: 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="b
asketball" 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.

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.mydoc.mydoc_definitions.basketball}}</td>
</tr>

<tr>
<td>Baseball</td>
<td>{{site.data.mydoc.mydoc_definitions.baseball}}</td>
</tr>

<tr>
<td>Football</td>
<td>{{site.data.mydoc.mydoc_definitions.football}}</td>
</tr>

<tr>
<td>Soccer</td>
<td>{{site.data.mydoc.mydoc_definitions.soccer}}</td>
</tr>
</tbody>
</table>
```

And here's the code:

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.

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.

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.

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.

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.)

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.

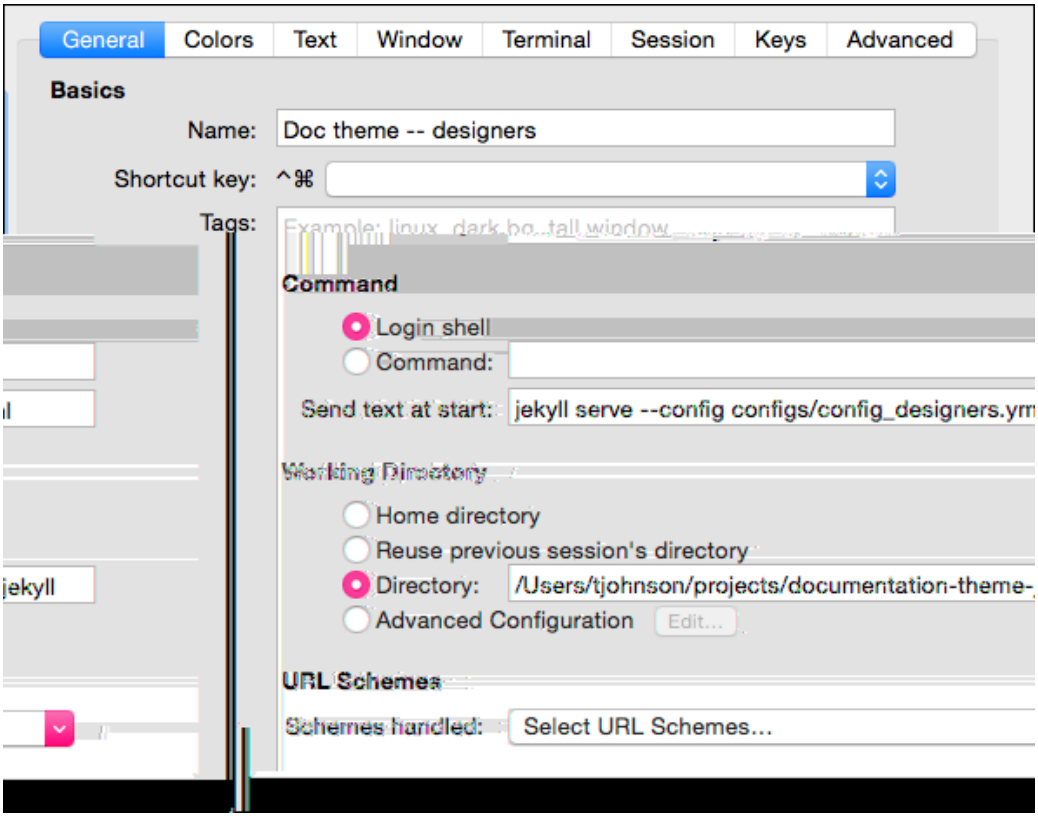
When you're working with tech docs, a lot of times you're single sourcing multiple outputs. It can be a hassle to fire up each one of these outputs using the build files containing the shell scripts. Instead, it's easier to configure iTerm with profiles that initiate the scripts.

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:



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**.

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.

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.

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.

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 O +       |
|      *.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/remoteserver/<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:

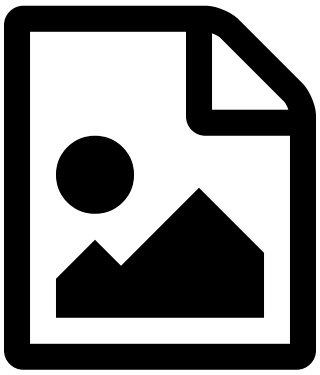
```
ssh <your_username>@remoteserver.com
```

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

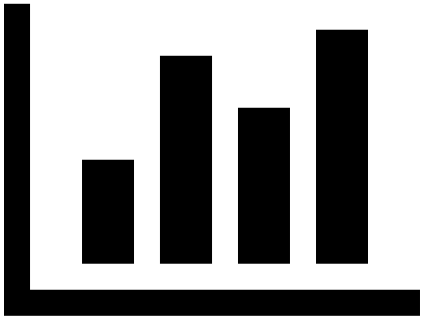
Now that you can connect without password prompts, you can use the scp scripts to transfer files to the server without password prompts. For example:

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

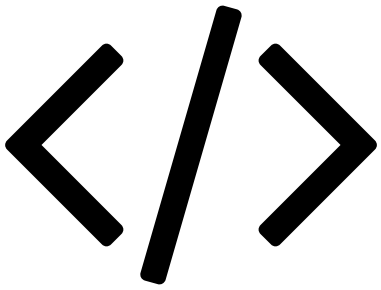
Summary: This shows a sample layout for a knowledge base. Each square could link to a tag archive page. In this example, font icons from Font Awesome are enlarged to a large size. You can also add captions below each icon.



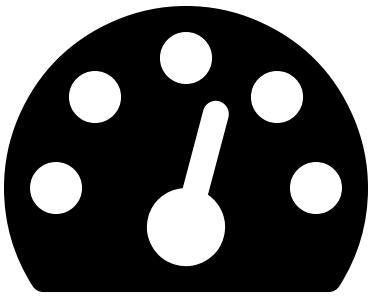
Getting Started



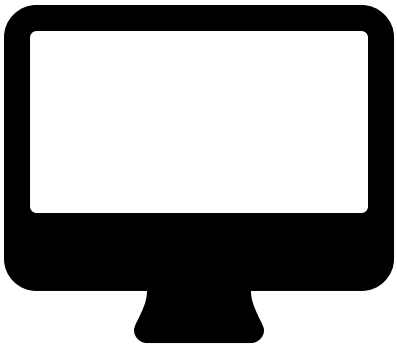
Navigation



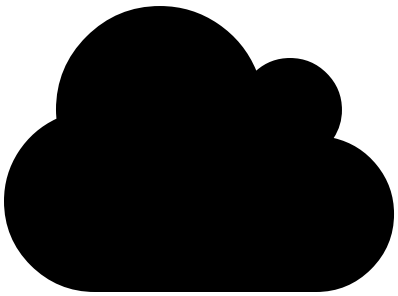
single_sourcing



Publishing



Special layouts



Formatting

If you don't want to link to a tag archive index, but instead want to list all pages that have a certain tag, you could use this code:

```
Getting started pages:
<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>
```

Getting started pages:

- 1. Build the default project (page 8)
- 2. Add a new project (page 12)
- 3. Decide on your project's attributes (page 15)
- 6. Configure the sidebar (page 29)
- About the theme author (page 0)
- Introduction (page 1)
- Pages (page 44)
- Sidebar Navigation (page 62)

- Support (page 0)
- Supported features (page 3)
- Troubleshooting (page 169)
- WebStorm Text Editor (page 50)

Summary: This page demonstrates how you the integration of a script called ScrollTo, which is used here to link definitions of a JSON code sample to a list of definitions for that particular term. The scenario here is that the JSON blocks are really long, with extensive nesting and subnesting, which makes it difficult for tables below the JSON to adequately explain the term in a usable way.

Note: The content on this page doesn't display well on PDF, but I included it anyway so you could see the problems this layout poses if you're including it in PDF.

```
{
  "apples" (page 157): "red fruit at the store",
  "bananas" (page 157): "yellow bananas in a bunch",
  "carrots" (page 157): "orange vegetables that grow in th
e ground",
  "dingbats" (page 157): "a type of character symbol on a c
omputer",
  "eggs" (page 157): "chickens lay them, and people eat the
m",
  "falafel" (page 158): "a Mediterranean sandwich consistin
g of lots of different stuff i don't know much about",
  "giraffe" (page 158): "tall animal, has purple tongue",
  "hippo" (page 158): "surprisingly dangerous amphibian",
  "igloo" (page 158): "an ice shelter made by eskimos",
  "jeep (page 158): "the only car that starts with a j",
  "kilt" (page 158): "something worn by scottish people, no
t a dress",
  "lamp" (page 158): "you use it to read by your bedside a
t night"
  "manifold" (page 158): "an intake mechanism on a car, lik
e a valve, i think",
  "octopus" (page 158): "eight tentacles, shoots ink, live
s in dark caves, very mysterious",
  "paranoia" (page 159): "the constant feeling that others
are out to get you, conspiring against your success",
  "qui" (page 159): "a life force that runs through your bo
dy",
  "radical" (page 159): "someone who opposes the status qu
o in major ways",
  "silly" (page 159): "how I feel writing this dummy copy",
  "taffy" (page 159): "the sweets children like the most an
d dentists hate the worst",
  "umbrella" (page 159): "an invention that has not had an
y advancements in 200 years",
  "vampire" (page 159): "a paranormal figure that is surpri
singly in vogue despite its basic nature",
  "washington" (page 159): "the place where tom was born",
  "xylophone" (page 160): "some kind of pinging instrument
used to sound chime-like notes",
  "yahoo" (page 160): "an expression of exuberance, said un
der breath when something works right",
  "zeta" (page 160): "the way british people pronounce z",
  "alpha" (page 160): "the original letter of the alphabe
t, which has since come to mean the first. however, i thin
k the original symbol of alpha is actually an ox. it is som
ewhat of a mystery to linquists as to the exact origin of t
he letter alpha, but it basically represents the dawn of th
```

```
e alphabet, which proved to be a huge step forward for huma
n thought and expression.",
  "beta" (page 160): "the period of time when something is
finished but undergoing testing by a group of people.",
  "cappa" (page 160): "how italians refer to their basebal
l caps",
  "dunno" (page 160): "informal expression for 'don't kno
w'"
  }
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer magna massa, euismod sed rutrum at, ullamcorper quis tellus. Vestibulum erat purus, aliquet sit amet pellentesque eget, tempus at ante. Nulla justo nisi, elementum nec nisi eget, consectetur varius tortor.

Curabitur quis nibh sed eros viverra tempus et quis lorem. Nulla convallis sit amet risus vitae rutrum. Nulla at faucibus lectus. Pellentesque tortor nisl, interdum ac quam non, egestas congue massa. Vestibulum non porttitor lacus. Nam tincidunt arcu lectus. Donec eget ornare neque, hendrerit ornare lectus. In ac pretium odio.

Vivamus pulvinar vestibulum pharetra. Vivamus vitae diam iaculis, posuere mi sed, dignissim massa. Nunc vitae aliquet urna. Proin sed pulvinar ex. Maecenas nisl lorem, rutrum sit amet hendrerit sed, posuere at odio. Sed consectetur semper tristique. Vivamus finibus varius felis at convallis. Fusce in dictum nunc.

Curabitur feugiat lorem eget elit ullamcorper tincidunt. In euismod diam aliquet tortor fermentum tempor. Fusce quam felis, commodo viverra orci vitae, scelerisque aliquet risus.

Duis est nunc, fringilla eu ligula et, varius dignissim dui. Vivamus in tellus vitae ipsum vehicula fermentum at congue tellus. Suspendisse fermentum, magna vitae aliquet sodales, tellus nisi rutrum arcu, vitae auctor dolor quam ac tellus. Cras posuere augue erat, in sagittis quam lacinia id.

Praesent auctor a enim non lacinia. Integer sodales aliquet mi vel dapibus. Donec consequat justo eget nisi lacinia, eu sodales ligula molestie. Sed sapien nulla, rhoncus at elementum a,

Aenean hendrerit mauris ipsum, non laoreet ipsum luctus vel. Curabitur tristique auctor elit ut pulvinar. Quisque arcu arcu, condimentum aliquam sodales nec, dignissim nec justo. Nunc tristique sem felis, pharetra euismod lorem volutpat sed. Ut porttitor metus sit amet elit rhoncus semper.

Quisque rhoncus cursus felis vel elementum. Vestibulum dignissim molestie tortor nec facilisis. Praesent a nibh condimentum, porta nulla egestas, auctor eros

Etiam hendrerit interdum tellus, at aliquet sapien egestas in. Aenean eu urna nisl. Cras vitae risus pharetra, elementum mauris nec, auctor lectus. Fusce pellentesque venenatis dictum. Proin at augue at mauris finibus semper ultricies sed eros.

Praesent pulvinar consequat posuere. Morbi egestas rhoncus felis, id fermentum metus lobortis in. Vestibulum nibh orci, euismod eget vestibulum nec, vehicula vitae tortor. Aenean ullamcorper enim nunc, eu auctor ligula auctor eget.

Etiam et arcu vel lacus aliquet lobortis in in massa. Nunc non mollis elit. Aenean accumsan orci quis risus aliquam, non gravida nulla molestie. Mauris pharetra libero et magna aliquam aliquam. Integer quis luctus dolor.

Fusce molestie finibus malesuada. Nullam ac egestas quam, id venenatis ligula. Pellentesque pulvinar elit et vestibulum fringilla. Cras volutpat sed quam ornare scelerisque. Vivamus volutpat ante pretium scelerisque tempus. Etiam venenatis tempor nisl dignissim sollicitudin. Curabitur ac risus vitae dolor pretium posuere vel vitae diam. Donec in odio arcu.

Vestibulum pretium condimentum commodo. Integer placerat leo non ipsum ultrices, ac convallis elit varius. Vestibulum ultricies, justo eu rutrum molestie, quam arcu euismod sapien, vel gravida ipsum nulla eget erat.

Nunc ac quam eu risus dictum sodales. Nam ac risus iaculis, aliquet sem eu, mollis mauris. Curabitur pretium facilisis orci ut lacinia. Sed fermentum leo a odio blandit rutrum. Phasellus at nibh vel odio interdum vulputate ac eget urna. Nam eu arcu dapibus, sodales ligula nec, volutpat ipsum. Suspendisse auctor tellus vitae libero euismod venenatis.

Sed molestie lobortis ante sit amet hendrerit. Sed pharetra nisi sed interdum pulvinar. Nunc efficitur erat non aliquam mattis. Sed id nisl mattis lacus vehicula volutpat vitae vel massa. Curabitur interdum velit odio, vitae sollicitudin nunc rutrum non.

Nunc commodo consectetur scelerisque. Proin fermentum ligula ac quam finibus tincidunt. Aenean venenatis nisi et semper semper. Nunc sodales velit ipsum, ac pellentesque augue placerat eu.

Praesent nec neque ac tellus sodales eleifend nec vel ipsum. Cras et semper risus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Integer mattis leo nisl, a tincidunt lectus tristique eget. Donec finibus lobortis viverra. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Vivamus egestas pulvinar odio non vehicula. Morbi malesuada leo eget nisl sagittis aliquet.

Nam molestie semper nulla et molestie. Ut facilisis, ipsum sed convallis posuere, mi mauris bibendum erat, nec egestas ipsum est nec dolor.

Etiam et metus congue, commodo libero et, accumsan sem. Aliquam erat volutpat. Quisque tincidunt, tortor non blandit ullamcorper, orci mauris dignissim augue, eget vehicula nulla justo sed dolor. Nunc ac urna quis nisi maximus pharetra in a mauris. Proin metus mi, venenatis vitae tristique sed, fermentum at purus. Aliquam erat volutpat. Maecenas efficitur sodales nibh, ac hendrerit felis facilisis et. Interdum et malesuada fames ac ante ipsum primis in faucibus.

Note: This was mostly an experiment to see if there was a better way to document a long JSON code example. I haven't actually used this approach in my own documentation.

Summary: This layout shows an example of a knowledge-base style navigation system, where there is no hierarchy, just groups of pages that have certain tags.

Note: The content on this page doesn't display well on PDF, but I included it anyway so you could see the problems this layout poses if you're including it in PDF.

All | Getting Started | Formatting | Publishing | Content types | Single Sourcing | Special Layouts

Getting started

If you're getting started with Jekyll, see the links in this section. It will take you from the beginning level to comfortable.

- Introduction (page 1)
- About the theme author (page 0)
- 2. Add a new project (page 12)
- 6. Configure the sidebar (page 29)
- 3. Decide on your project's attributes (page 15)
- 1. Build the default project (page 8)

Content types

This section lists different content types and how to work with them.

- Collections (page 60)
- Generating PDFs (page 115)
- Help APIs and UI tooltips (page 130)
- Pages (page 44)
- Series (page 71)

Formatting

- Pages (page 44)
- Sidebar Navigation (page 62)
- Support (page 0)
- Supported features (page 3)
- Troubleshooting (page 169)
- WebStorm Text Editor (page 50)

These topics get into formatting syntax, such as images and tables, that you'll use on each of your pages:

- Tooltips (page 74)
- Alerts (page 75)
- Glossary layout (page 166)
- Links (page 89)
- Icons (page 79)
- Images (page 85)
- Labels (page 88)
- Navtabs (page 94)
- Pages (page 44)
- Syntax highlighting (page 105)
- Tables (page 101)
- Video embeds (page 97)

Single Sourcing

These topics cover strategies for single_sourcing. Single sourcing refers to strategies for re-using the same source in different outputs for different audiences or purposes.

- Conditional logic (page 53)
- 4. Set the configuration options (page 17)
- Content reuse (page 58)

Publishing

When you're building, publishing, and deploying your Jekyll site, you might find these topics helpful.

- Build arguments (page 108)
- 10. Configure the build scripts (page 38)
- 4. Set the configuration options (page 17)

- Excluding files (page 127)
- Generating PDFs (page 115)
- Help APIs and UI tooltips (page 130)

- Generating PDFs (page 115)
- Help APIs and UI tooltips (page 130)
- iTerm profiles (page 144)
- Link validation (page 112)
-

Note: This was mostly an experiment to see if I could break away from the hierarchical TOC and provide a different way of arranging the content. However, this layout is somewhat problematic because it doesn't allow you to browse other navigation options on the side while viewing a topic.

Summary: You can use an accordion-layout that takes advantage of Bootstrap styling. This is useful for an FAQ page.

If you want to use an FAQ format, use the syntax shown on the faq.html page. Rather than including code samples here (which are bulky with a lot of nested `div` tags), just look at the source in the mydoc_faq.html theme file.

Lorem ipsum dolor sit amet, consectetur adipiscing elit?

Curabitur eget leo at velit imperdiet varius. In eu ipsum vitae velit congue iaculis vitae at risus?

Aenean consequat lorem ut felis ullamcorper?

Lorem ipsum dolor sit amet, consectetur adipiscing elit?

Curabitur eget leo at velit imperdiet varius. In eu ipsum vitae velit congue iaculis vitae at risus?

Aenean consequat lorem ut felis ullamcorper?

Lorem ipsum dolor sit amet, consectetur adipiscing elit?

Curabitur eget leo at velit imperdiet varius. In eu ipsum vitae velit congue iaculis vitae at risus?

Aenean consequat lorem ut felis ullamcorper?

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></dd>

<dt id="gratuitous">gratuitous</dt>
<dd></dd>

<dt id="haughty">haughty</dt>
<dd></dd>

<dt id="gratuitous">gratuitous</dt>
<dd></dd>

<dt id="impertinent">impertinent</dt>
<dd></dd>

<dt id="intrepid">intrepid</dt>
<dd></dd>

</dl>
```

Here's what that looks like:

fractious

gratuitous

haughty

gratuitous

impertinent

intrepid

The glossary works well as a link in the top navigation bar.

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

gratuitous

haughty

gratuitous

impertinent

intrepid

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.

Summary: This page lists common errors and the steps needed to troubleshoot them.

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/conditions.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.

> **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.