Jekyll Documentation Theme for Designers

version 3.0

Last generated: September 13, 2015

Company logo

Table of Contents

Getting started

Authoring

Navigation

Formatting

Single-sourcing

Publishing

Special layouts

Tag archives

Introduction

Overview

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

Survey of features

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

- Bootstrap framework

- Sidebar with page hierarchy and advanced toc

- PDF generation (with Prince XML utility)

- Notes, tips, and warning information notes

- Tags

- Single sourced outputs

- Emphasis on pages, not posts

- Relative (rather than absolute) link structure

I'm using this theme for my documentation projects, building about 15 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 .

Getting started

To get started, see these three topics:

PDF Download

If you would like to download this help file as a PDF, you can do so here. The PDF most of the same content as the online help, except that some elements (such as video or special layouts) don't translate the the print medium, so they're excluded.

⬇ PDF Download

The PDF contains a timestamp in the header indicating when it was last generated. If you download a PDF, keep in mind that it may go out of date quickly. Always compare your PDF timestamp against the online help timestamp (which you can find in the footer).

Getting started with this theme

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).

Step 1: Set up the prerequisites

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

- **Mac computer (recommended)**. If you have a PC, you need to see Jekyll on Windows (http://jekyllrb.com/docs/windows/). Make sure you can get Jekyll working on Windows before proceeding.

- **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.

- **Jekyllrb (http://jekyllrb.com/)**. To install: `gem install jekyll`. Type `which jekyll` to confirm that Jekyll is installed.

- **Text editor** (some examples: Sublime Text, Atom, WebStorm, IntelliJ)

- **iTerm (http://iterm.sourceforge.net/)** - Optional but recommended instead of Terminal.

- **pygments (http://pygments.org/download/)** - Pygments handles syntax highlighting. In my experiments, the Pygments highlighter seemed better than the default rouge highlighter. To install Pygments, you will need Python installed. (If you don't install pygments, you'll get an error when you build the theme.) To check if Python is installed, type `which python`. To install Pygments: `gem install pygments.rb`. If you want to use `rouge` instead, change `pygments` to `rouge` in the configuration files.

Step 2: Build the theme

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. Note, however, that you won't be using the pull command to update the theme since you'll be customizing it with your own content and won't want to overwrite those customizations, so there isn't a need to clone it.

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.

    ◦ 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.

    ◦ You can build all the outputs in your configs directory by running the doc_multibuild_web.sh file in the root directory.

> ☑ **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 site 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 involve this requirement.

There are four configuration files in this project: config_writer.yml and config_designer.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. However, for the writers output, I've conditionally excluded more lengthy explanations about how the theme works. The idea is that writers just want to create and publish content; in contrast, designers want to understand and modify the theme itself. Also, the configuration files use different themes.

3. Build the writer's output:

```
jekyll serve --config configs/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.

4. Press **Ctrl+C** in Terminal to shut down the writer's output.

5. Build the designers output:

```
jekyll serve --config configs/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).

6. Press **Ctrl+C** in Terminal to shut down the designer's output.

7. Build both themes by running the following command:

```
. doc_multibuild_web.sh
```

The themes build in the ../doc_designers and ../doc_writers folders. Use finder and browse to one level above where you installed the project (probably username/projects).

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 build 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, try to solve them before moving on. See for more information.

> ☑ **Tip:** You can set up profiles in iTerm to initiate all your builds with one selection. See iTerm profiles (page 125) for details.

```
More information about building the PDF versions is provided i
n <a href='doc_generating_pdfs.html'>Generating PDFs</a>.
```

Questions

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.

Setting configuration options

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

Importance of Configuration File

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 all the settings and other details unique to that site output, such as variables, titles, output directories, build folders, and more.

> ❶ **Warning:** This theme is coded to look for specific values set by the configuration file. If something isn't working correctly, check to make sure that you have the configuration values that are defined here.

Configuration file options

Some of the options you can set in the configuration file determine theme settings.

Note that 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, the values in these tables are used to control different aspects of the theme and are not arbitrary key-value pairs.

Configuration settings for web outputs

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 _includes/conditions.html file. Therefore 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.
audience	Required	The audience for the output. Each entry in _data/sidebar_doc.yml and _data/topnav_doc.yml needs to have an audience attribute that matches the value here in order for the sidebar or topnav item to be included.
platform	Required	The platform for the output. See additional information in audience.
product	Required	The product for the output. See additional information in audience.
version	Required	The version for the output. See additional information in audience.
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, such as /Applications/XAMPP/xamppfiles/htdocs/myfolder.

FIELD	REQUIRED?	DESCRIPTION
project_file_name	Required	The shortname for your project that you preface each file name with (for example, `doc`). This label is used to specify the prefix for the tag archives files (which are named with titles such as `doc_tag-formatting.html`). The raw code in the theme is `<a href="{{site.project_file_name}}_tag-{{tag}}.html">` The `{{site.project_file_name}}` field renders as `doc` in the sample theme. The `{{tag}}` is populated by a "for" loop through the tags property specified in page frontmatter.
sidebar_tagline	Optional	Appears above the sidebar. Usually you put some term related to the site specific build, such as the audience. In the sample theme files, the taglines are "writers" and "designers."
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 "3.0."
topnav_title	Required	Appears next to the home button in the top nav bar. In the sample theme files, the topnav_title is "Jekyll Documentation Theme."
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 -- {audience}."

FIELD	REQUIRED?	DESCRIPTION
site_title	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 the "page name	homepage title."
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	Gets configured as the email address in the Send Feedback button in the top navigation bar.	
sidebar_accordion	Optional	Boolean. The default value is true. Whether you want the navigation sidebar to use the accordion effect or not. The accordion effect means when you expand a section, the other sections automatically collapse. If you put `false`, then you can expand multiple sections at once. At the bottom of the navigation sidebar, two links will appear: Collapse All and Expand All.
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.

FIELD	REQUIRED?	DESCRIPTION
markdown	Required	The processor to use for Markdown. This is a Jekyll-specific setting. Use `redcarpet`. Another option is `kramdown`. However, my examples will follow redcarpet.
redcarpet	Required	Extensions used with redcarpet. You can read more about them by searching for redcarpet extensions online.
highlighter	Optional	The syntax highlighter used. Use `pygments` because it's required you're publishing on Github Pages. You will need to install Pygments (http://pygments.org/download/) on your machine or else you will see an error. Pygments is based on Python. If you run into build errors and aren't publishing on Github Pages, `rouge` is also an option.
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.
defaults	Optional	Here you can set default values for frontmatter based on the content type (page, post, or collection).
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.
print	Optional	Boolean. Whether this build is a print build or not. This setting allows you to run conditions in your content such as {% if site.print == true %} do this... {% endif %} .
suffix	Optional	If you publish on Salesforce's site.com, you have to add index.html to the permalink or else the page won't render. If you add `suffix: index.html` in your config file, this suffix will be appended in the homepage URL. If you're not publishing to Salesforce, don't add this property to your configuration file.

Where to store configuration files

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

The conditional attributes

Each configuration file must specify values for the conditional attributes:

- project

- product

- platform

- audience

- version

The sidebar.html and topnav.html files apply conditional logic based on the values for these conditional attributes.

For example, you will see this kind of logic in the sidebar and topnav files:

```
  {% if item.audience contains audience and item.product contain
 s product and item.platform contains platform and item.version
 contains version and item.web != false %}
```

If all of these conditions are met, then the item will qualify to be included in the sidebar or top navigation file. That is why each item in the sidebar_doc.yml or topnav_doc.yml file includes similar properties to match:

```
- title: Pages
  url: /doc_pages.html
  audience: writers, designers
  platform: all
  product: all
  version: all
```

The file in _includes/custom/conditions.html contains a project setting and also assigns general names for each of these specific values. This way the same theme files can be used interchangeably depending on the assignments, whose values are specified in the configuration file.

It's a little complicated to describe, but it works. Once you configure your project correctly, you don't even think about how the theme is processing all of this on the backend.

Configuration settings for PDF output

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,configs/config_writers_pdf.yml
```

First Jekyll will read the config_writers.yml file, and then Jekyll will read the config_writers_pdf.yml file.

More detail about generating PDFs is provided in Generating PDFs (page 98), 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. (You might be able to remove this first step by doing more coding, but I found it easier just to strip out components I didn't want included and make other adjustments.)

FIELD	REQUIRED?	DESCRIPTION
destination	Where the PDF web version should be served so that Prince XML can find it. By default, this is in ../doc_designers-pdf, so just one level above where your project is.	

FIELD	REQUIRED?	DESCRIPTION
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. (My attempts to use local file paths didn't work.)
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.
print		A boolean so that you can construct conditional statements in your content to check whether print is true or not. This setting can help you filter out content that doesn't fit well into a PDF (such as dynamic web elements).
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 3.0."

FIELD	REQUIRED?	DESCRIPTION
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.	

Customizing the theme

> **Summary:** You start customizing the theme by gutting the existing content in this theme and replacing it with your own content. Start with the configuration files, then customize the data files, and add your own markdown pages in the root directory.

About customizing the theme

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. Follow these steps to customize the theme with your own content.

> ⚠ **Important:** In these instructions, I'll assume your project's name is "acme." I'll also assume you have two audiences you're building your acme project for: marketers and developers.

To customize the theme:

1. In the theme's root directory, rename config_writer.yml to config_marketer.yml and customize all the values inside that file based on the instructions in Setting configuration options (page 7). Do the same with config_designer.yml (changing it to config_developer.yml) and continue to clone and customize the config file for other audiences you need.

   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.

2. Make similar customizations to the PDF configuration files. You will later use these files when you create PDFs.

   > ☑ **Tip:** As you customize the config files, make the port values unique so that you don't run into "Address already in use" issues when you build multiple sites and want to preview them at the same time.

3. In the _includes/custom directory, open conditions.html and customize the values there specific to your outputs. (Basically, replace `writer` with `developer`, and `designer` with `marketer`.)

   The conditions.html file is used to apply different requirements to the sidebar and other files. The conditions.html file is included in various parts of the theme — the sidebar.html, the topnav.html, and some of the print files. *conditions.html is sort of the brains of the theme.* If you don't have a specific value for audience, version, platform, or product, just put `all`.

4. Remove the pages that begin with "doc_" in the root directory, and then add your own pages here. Leave all the files flat in the root directory.

   If you nest files inside folders, you'll create problems for the links and the theme will break. Yes, this will result in a lot of files in the root directory, but you can get around this issue with some viewing strategies in your text editor.

   For example, with WebStorm, if you press **Shift** twice and type the file name you want, the editor finds it. I usually have the preview mode open in another browser and navigate the content that way. When I want to edit a specific file, I copy the filename path from the preview browser, press **Shift** twice, and then it opens. You can also create a favorites section that just shows files you've added to Favorites (an option in the context menu).

5. Inside _data, open sidebar_doc.yml and topnav_doc.yml and customize the navigation.

   > ❶ **Warning:** Don't mess up the spacing or change any of the YML level names or the site or sidebar won't appear. Each new YML level is indented with two spaces. Sometimes getting this spacing right is tricky. I recommend you save the sample template here that shows the various levels, and then just copy and paste the levels where you need them. YML is very picky and it can be frustrating sorting out spacing and level issues.

6. In the root directory, customize the index.md file. This file will be the homepage for all of your projects.

   Use conditional tags (for example, `{% if site.project == "writers" %} ... {% endif %}`) to change the content for different builds of your site. Store the content of the homepage in your _includes/custom/{project_name} folder. See for more information on applying conditions.

7. In the _includes folder, open footer.html and customize the content
    (namely the footer image). If you have different footers for different
    outputs, use conditional tags as you did with index.md.

8. Build your site with a command such as
    `jekyll serve --config configs/config_writers.yml` etc., and
    preview it at the URLs provided.

> **Publishing to web hosts:** If you have multiple outputs, you probably don't
> want to use Github Pages to publish your site, since Github Pages looks
> for a _config.yml file in the root directory and uses that to build a site in
> the gh-pages branch of your repository. You can't instruct Github pages to
> look in another directory for the right configuration file. Instead, you'll
> probably have a better experience publishing to a Amazon Web Services
> S3 bucket or some other web host. See Pushing builds to server (page
> 127) for more information.

Supported features

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

Before you get into exploring Jekyll as a potential platform for help content, you may be wondering if it supports some basic features. 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.

FEATURES	SUPPORTED	NOTES
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.
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.

FEATURES	SUPPORTED	NOTES
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.
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.

FEATURES	SUPPORTED	NOTES
CMS interface	No	Unlike with WordPress, you don't log into an interface and navigate to your files. You work with text files and preview the site dynamically in your browser. Don't worry -- this is part of the simplicy that makes Jekyll awesome. I recommend using WebStorm as your text editor.
WYSIWYG interface	No, 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 plugins	Yes	You can use any jQuery plugins you and other JavaScript, CMS, or templating tools. However, note that if you use Ruby plugins, you can't directly host the source files on Github Pages because Github Pages doesn't allow Ruby plugins. Instead, you can just push your output to any web server. If you're not planning to use Github Pages, there are no restrictions on any plugins of any sort. Jekyll makes it super easy to integrate every kind of plugin imaginable. This theme doesn't actually use any plugins, so you can publish on Github if you want.
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.

FEATURES	SUPPORTED	NOTES
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.
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.

Pages

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.

Where to author content

Use a text editor such as Sublime Text, WebStorm, IntelliJ, or Atom to create pages.

My preference is IntelliJ/WebStorm, since it will treat all files in your 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.

Page names and excluding files from outputs

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, doc_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:

- doc_*
- doc_writers_*

These wildcards will exclude every match after the * .

Frontmatter

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: August 12, 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.

FRONTMATTER	REQUIRED?	DESCRIPTION
summary	Optional	A 1-2 word sentence summarizing the content on the page. This gets formatted into the summary section in the page layout. Adding summaries is a key way to make your content more scannable by users (check out Jakob Nielsen's site (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.
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?

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).

Colons in page titles

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

Saving pages as drafts

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

> ☑ **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 32) for details.

Markdown or HTML format

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

If you use Markdown, you can also include HTML formatting where needed. But not vice versa — 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.

Where to save pages

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.

Github-flavored Markdown

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.

Automatic mini-TOCs

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.

Specify a particular page layout

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

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

Comments

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

Posts

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.

Custom keyboard shortcuts

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

For example, 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).

WebStorm Text editor

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

About text editors and WebStorm

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

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

Remove unnecessary plugins

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

Add the Markdown Support plugin

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

Learn a few key commands

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.

Identifying changed files

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

Creating file templates

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

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

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

2. In the upper-left corner of the dialog box that appears, click the **+** button to create a new template.

3. Name it something like Jekyll page. Insert the frontmatter you want, and save it.

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

Disable pair quotes

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

Series pages

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

Using series for pages

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

1. Create the series button

First create an include that contains your series button:

```
<div class="seriesContext">
    <div class="btn-group">
        <button type="button" data-toggle="dropdown" class="bt
n btn-primary dropdown-toggle">Series Demo <span class="care
t"></span></button>
        <ol class="dropdown-menu">
            {% assign pages = site.pages | sort:"weight"  %}
            {% for p in pages %}
            {% if p.series == "ACME series" %}
            {% if p.url == page.url %}
            <li class="active"> → {{p.weight}}. {{p.title}}</l
i>
            {% else %}
            <li>
                <a href="{{p.url | replace: '/',''}}">{{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 folder as something like series_acme.html.

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

2. Create the "next" include

This will be the next button at the bottom of the page:

```
<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 | replace: '/',''}}"><button type="butto
n" class="btn btn-primary">Next: {{p.weight}}  {{p.title}}</but
ton></a>
    {% endif %}
    {% endif %}
    {% endfor %}
</p>
```

Change "acme" to the name of your series.

Save this in your _includes folder as series_acme_next.html.

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

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

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

With 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"`.

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

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

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

{% include custom/doc/series_acme.html %}

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

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

Changing the series drop-down color

The Bootstrap menu uses the `primary` class for styling. If you change this class in your theme, the Bootstrap menu should automatically change color as well. You can also just use another Bootstrap class in your button code. Instead of `btn-primary`, use `btn-info` or `btn-warning`. See Labels (page 69) for more Bootstrap button classes.

Collections

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

What are collections

Collections are custom content types different from pages and posts. You might create a collection if you want to treat a specific set of articles in a unique way, such as with a custom layout or listing. For more detail on collections, see Ben Balter's explanation of collections here (http://ben.balter.com/2015/02/20/jekyll-collections/).

Create a collection

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

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

In this example, "tooltips"" is the name of the collection.

Interacting with collections

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

See Collections in the Jekyll documentation (http://jekyllrb.com/docs/collections/) for more information.

How to use collections

I haven't found a huge use for collections in normal documentation. However, I did find a use for collections in generating a tooltip file that would be used for delivering tooltips to a user interface from text files in the documentation. See {"title"=>"Help API and UI tooltips", "url"=>"doc_help_api.html", "link"=>"Help API and UI tooltips (page 112)"} for details.

Sidebar navigation

> **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.

Sidebar overview

To configure the sidebar, edit the values in the _data/sidebar_doc.yml file. Follow the example in this theme. Note that YML spacing is picky. Each new level is two spaces indented. I usually just keep a template that shows all three levels and then copy and paste from it as needed.

Sidebar levels

You can add three levels of nesting in the sidebar nav. For example, three levels looks like this:

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

You can't add more than three levels. In general, it's a best practice not to create more than three 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.

External links

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>
```

How it works

The values in the sidebar_doc.yml file are coded to match the logic in includes/sidebar.html.

Each of the items in the sidebar needs to have the attributes shown here:

```
- title: Getting started
  url: /doc_getting_started.html
  audience: writers, designers
  platform: all
  product: all
  version: all
```

The audience, platform, product, and version are specified in the includes/custom/conditions.html file:

```
{% if site.project == "doc_designers" %}
{% assign audience = "designers" %}
{% assign sidebar = site.data.sidebar_doc.entries %}
{% assign topnav = site.data.topnav_doc.topnav %}
{% assign topnav_dropdowns = site.data.topnav_doc.topnav_dropdo
wns %}
{% assign version = "all" %}
{% assign product = "all" %}
{% assign platform = "all" %}
{% assign link = "custom/doc/links_doc.html" %}
{% assign projectTags = site.data.tags_doc.allowed-tags %}
{% assign searchGroup = "doc" %}
{% endif %}
```

Additionally, note how some assignments are set here as well. The conditions.html file set things like `sidebar = site.data.sidebar_doc.entries` .

When the sidebar.html file runs the logic, it includes these statements:

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

{% for entry in sidebar %}
...
```

The assignment of the `sidebar` value through the conditions.html file means this is really what's happening:

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

{% for entry in site.data.sidebar_doc.entries %}
```

Since different projects will use different data files, I had to make the logic run using the standard `sidebar` variable, but change the meaning of that variable based on the project.

Sidebar accordion

The configuration file (configs/config_writers.yml) file includes a value
(`sidebar_accordion`) that lets you choose whether to use the accordion feature
in the sidebar or not. The accordion feature collapses other sections when a
section is opened, which conserves space on the screen. Put `true` or `false` for
the value.

Sidebar fixed or moving

If you scroll up, the sidebar usually remains fixed in place. This is so that users
can keep the context of the table of contents in place while they move down the
page.

However, if the user's viewport is really small, you don't want the TOC to remain
fixed in place because expanding one section may extend past what is visible. In
the js/customscripts.js file, I inserted some logic to check the viewport size.

```
$( document ).ready(function() {

    //
    var h = $(window).height();
    console.log (h);
    if (h > 800) {
        $( "#mysidebar" ).attr("class", "nav affix");
    }
    // activate tooltips. although this is a bootstrap js funct
ion, it must be activated this way in your theme.
    $('[data-toggle="tooltip"]').tooltip({
        placement : 'top'
    });

});
```

The script says, if the height of the viewport is greater than 800px, then insert
`affix` class, which makes the nav bar fixed as your scroll. If you have a lot of
nav items, this fixed position may not work for you.

For example, if your sidebar has a lot of items and the fixed position makes it so the user can't see all the items when expanded, you may want to adjust the height. If viewing the sidebar is ap roblem, increase the height value from `800` to `1000` or more.

Navgoco foundation

The sidebar uses the Navgoco jQuery plugin (https://github.com/tefra/navgoco) as its basis. Why not use Bootstrap? Navgoco provides a few features that I couldn't find in Bootstrap:

- Navgoco sets a cookie to remember the user's position in the sidebar. If you refresh the page, the cookie allows the plugin to remember the state.

- Navgoco inserts an `active` class based on the navigation option that's open. This is essential for keeping the accordion open.

- Navgoco includes the expand and collapse features of a sidebar.

In short, the sidebar has some complex logic here. I've integrated Navgoco's features with the sidebar.html and sidebar_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.)

Highlighting the active item

Here's how the highlighting of the currently viewed page works. In the sidebar.html file, you'll see this:

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

The `page.url` is a universal Jekyll tag that returns the end path of the page, prepended with `/`. For example, `/sample.html`. If this `page.url` matches the `item.url` specified in the sidebar_doc.yml file, then an `active` class gets applied.

Note that I've added a filter to the item.url in the sidebar.html file:

```
{{item.url | replace: "/",""}}
```

Ideally, in the sidebar_doc.yml file, you could just write the URL you want to go to: `sample.html` instead of `/sample.html` . However, page.url always returns the forward slash before the URL. In order to match the page.url with the item.url, you have to use the forward slash before item.url in the doc_sidebar.yml file.

However, if you set up your relative link as `/sample.html` instead of `sample.html` , then the browser will go to the directory root instead of any baseurl.

For example, if your site is deployed at http://mydomain.com/doc/, then going to `/sample.html` in the link will take you to `http://mydomain.com/sample.html` instead of `http://mydomain.com/doc/sample.html` .

In contrast, going to `sample.html` in the link will take you to `http://mydomain.com/doc/sample.html` . Hence the filter to remove the forward slash in the link.

That logic handles the highlighting of the active item, but in order for the sidebar to remain open, the parent level needs to also have the `active` class. To accomplish this, I inserted this script below the sidebar code in the sidebar.html file:

```
<script>$("li.active").parents('li').toggleClass("active");</script>
```

This script has to come *after* the sidebar code. Otherwise, if placed inside customscripts.js, the script runs before the sidebar code runs and the class never gets inserted.

Top navigation

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.

Changing the top navigation

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

- `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.

The Feedback email

If you click the Feedback link, 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. This 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 with this property: `site.feedback_email`.

Custom Menu

It's common to publish multiple sites. If you want to link them together, you could simply list links to the other doc sites in a drop-down menu configured in the topnav_dropdowns section in the topnav_doc.yml file. However, suppose you want to do something more fancy.

Included in the topnav.html file is an include to /doc/customMenu.html. The code in customMenu.html is as follows:

```
<li ><a href=" ../doc_writers/doc_top_navigation.html">Writer d
ocs</a></li>

<li class="dropdownActive"><a href=" ../doc_designers/doc_top_n
avigation.html">Designer docs</a></li>
```

> ❶ **Note:** In the theme, the link to the customMenu.html include in the _includes/topnav.html file is currently commented out. This is because this feature only works when you have multiple outputs hosted on the same server. Github Pages, where I'm publishing this theme, allows only one output per Github Pages directory. So rather than removing this customMenu feature from the theme, I've just commented it out so that it won't appear broken in the demo.

The current doc site is highlighted. If you select another doc site, the site switches to that doc site and goes to the same page on that doc site. This way, if you have a task such as "Configuring the license" in several different programming languages, users can switch to other programming languages to see the same page.

You need to have both the designers and writers sites deployed on a web server to see this in action. Once deployed, browse to any page in the navigation. Then go to the **Custom Menu** and select the **Writers** site. You'll go to the exact same page but on the Writers site.

If your current page doesn't have an equivalent in your other outputs, then put this in the frontmatter of the page:

```
switch: false
```

This Custom Menu may not be something you want, and if so, just remove the include from the sidebar.html file. But if you're outputting multiple sites, it may be something valuable.

Pop-out link

The top navigation bar also has an include to frameescape.html. If the site is embedded inside a frame, a link on the top navigation bar appears that says Pop-out, and it will open the site in a new window.

In most cases, you'll want to simply remove this include. I added this because some of my doc sites are delivered through a Salesforce Community and are embedded inside another page in a small area. This pop-out link is a way of liberating the site from these embedded page scenarios. If your site isn't embedded in an iframe, the Pop-out link is removed.

Tags

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

Add a tag to a page

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

```
---
title: 2.0 Release Notes
permalink: /release_notes_2_0/
tags: [formatting, single-sourcing]
---
```

Tags overview

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

> ❶ **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 doc_tag-{tagname}.html folder. This theme doesn't auto-create tag archive pages.

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

Setting up tags

Tags have a few components.

1. 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: doc_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 %}
   ```

```
<div class="alert alert-info" role="alert"><i class="fa
fa-info-circle"></i> <b>Note: </b>In the \_includes fold
er, there's a taglogic.html file. This file (included i
n each tag archive file) has common logic for getting th
e 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 &mdash; just leave it
there because the tag archive pages reference it.</div>
```

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="{{site.project}}_tag-{{tag}}.html"><button
type="button" class="btn btn-info navbar-btn cursorNor
m">{{page.tagName}}{{tag}}</button></a>{% unless forloo
p.last %}{% endunless%}
    {% endif %}
    {% endfor %}
    {% endif %}
</div>
```

   Here's an example of what the code does:

   Tags:

   navigation

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 for more options on button class names.

Retrieving pages for a specific tag

If you want to retrieve pages outside of a particular doc_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 | replace: '/',''}}">{{page.titl
e}}</a></li>
{% endif %}
{% endfor %}
{% endfor %}
</ul>
```

Here's how that code renders:

Getting started pages:

- About this theme (page 0)

- Customizing the theme (page 16)

- Getting started with this theme (page 3)

- Pages (page 25)

- Support (page 0)

- Supported features (page 19)

- Troubleshooting (page 0)

- WebStorm Text editor (page 32)

- Introduction (page 1)

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

Here's how that code renders:

Getting started pages:

- About this theme (page 0)

- Customizing the theme (page 16)

- Getting started with this theme (page 3)

- Introduction (page 1)

- Pages (page 25)

- Support (page 0)

- Supported features (page 19)

- Troubleshooting (page 0)

- WebStorm Text editor (page 32)

Efficiency

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

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

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

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

Empty tags?

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

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

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

Remembering the right tags

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

See WebStorm Text Editor (page 32) for tips on creating file templates in WebStorm.

Tooltips

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

Creating tooltips

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

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

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

This renders to the following:

Jekyll is my favorite tool for building websites.

Alerts

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

About alerts

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

Alerts

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

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

The following demonstrate the formatting associated with each alert.

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

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

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

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

Callouts

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

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

The following demonstrate the formatting for each callout.

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

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

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

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

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

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

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

Blast a warning to users

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

Using Markdown inside of notes

You can't use Markdown formatting inside alerts. This is because the alerts leverage HTML, and you can't use Markdown inside of HTML tags.

Icons

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

Font icon options

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

See Font Awesome icons available

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

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

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

```
<i class="fa fa-camera-retro"></i> normal size (1x)
<i class="fa fa-camera-retro fa-lg"></i> fa-lg
<i class="fa fa-camera-retro fa-2x"></i> fa-2x
<i class="fa fa-camera-retro fa-3x"></i> fa-3x
<i class="fa fa-camera-retro fa-4x"></i> fa-4x
<i class="fa fa-camera-retro fa-5x"></i> fa-5x
```
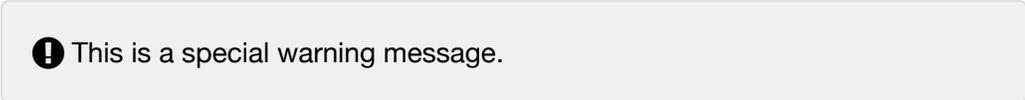
Here's what they render to:

1x fa-lg fa-2x fa-3x fa-4x fa-5x

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

```
<div class="alert alert-danger" role="alert"><i class="fa fa-ex
clamation-circle"></i> <b>Warning: </b>This is a special warnin
g message.
```

Here's the result:

> ❗ This is a special warning message.

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

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

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

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

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

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

Here's the result:

> ⓘ **Note:** Add your note here.

> ☑ **Tip:** Here's my tip.

> ⚠ **Important:** This information is very important.

> ⊖ **Warning:** If you overlook this, you may die.

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

Creating your own combinations

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

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

And the result:

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

Grab the right class name from the Font Awesome library (http://fortawesome.github.io/Font-Awesome/icons/) and then implement it by following the pattern shown previously.

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

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

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

Glyphicon icons available

Glyphicons work similarly to Font Awesome. Go to the Glyphicons library
(http://getbootstrap.com/components/#glyphicons) to see the icons available.

Although the Glyphicon Halflings library doesn't provide the scalable classes like
Font Awesome, there's a StackOverflow trick (http://stackoverflow.com/
questions/24960201/how-do-i-make-glyphicons-bigger-change-size) to make the
icons behave in a similar way. This theme's stylesheet (customstyles.css) includes
the following to the stylesheet:

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

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

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

And here's the result:

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

Here's another example:

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

And magnified:

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

⊕

You can also put glyphicons inside other elements:

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

❗ **Error:** Enter a valid email address

Callouts

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

```
<div class="bs-callout bs-callout-info">
 This is a special info message. This is a special info messag
e. This is a special info message. This is a special info messa
ge. This is a special info message. This is a special info mess
age. This is a special info message. This is a special info mes
sage. This is a special info message. </div>
```

❓ 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 56).

Images

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

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

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

```
<img title="my sample page" src="images/jekyll.png" />
```

And the result:

Here's the same Markdown syntax:

```
![My sample page](images/jekyll.png)
```
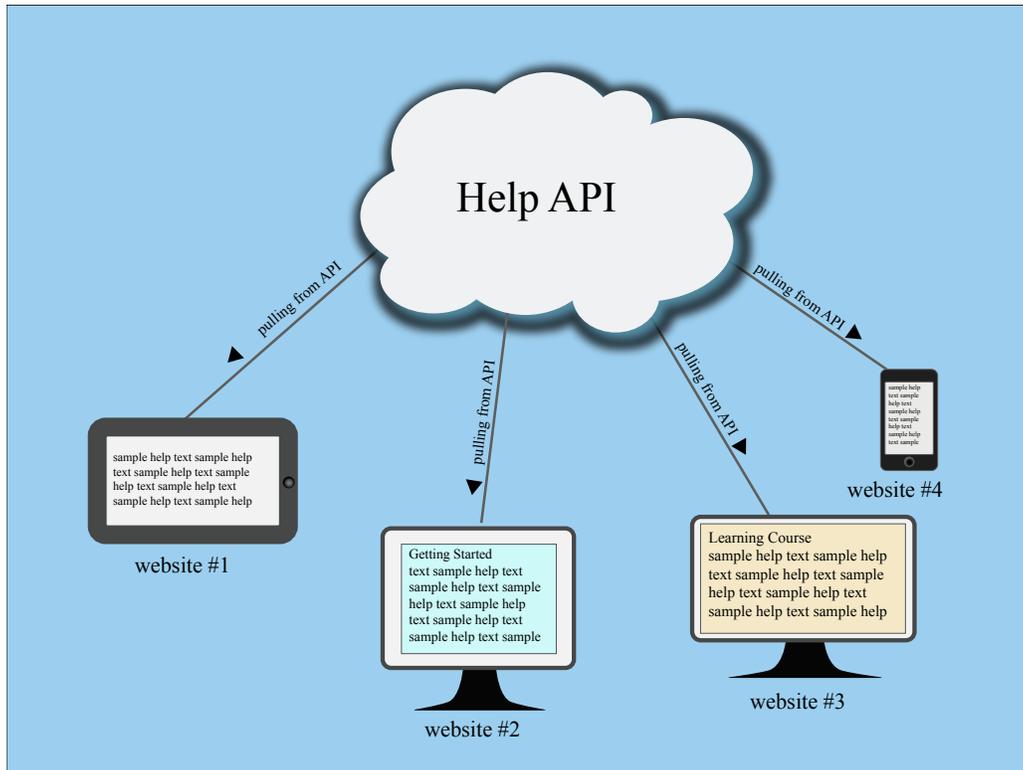
And the result:

SVG Images

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

```
<img src="images/dicloud/dicloud_architecture.svg" style="max-w
idth: 700px;" />
```
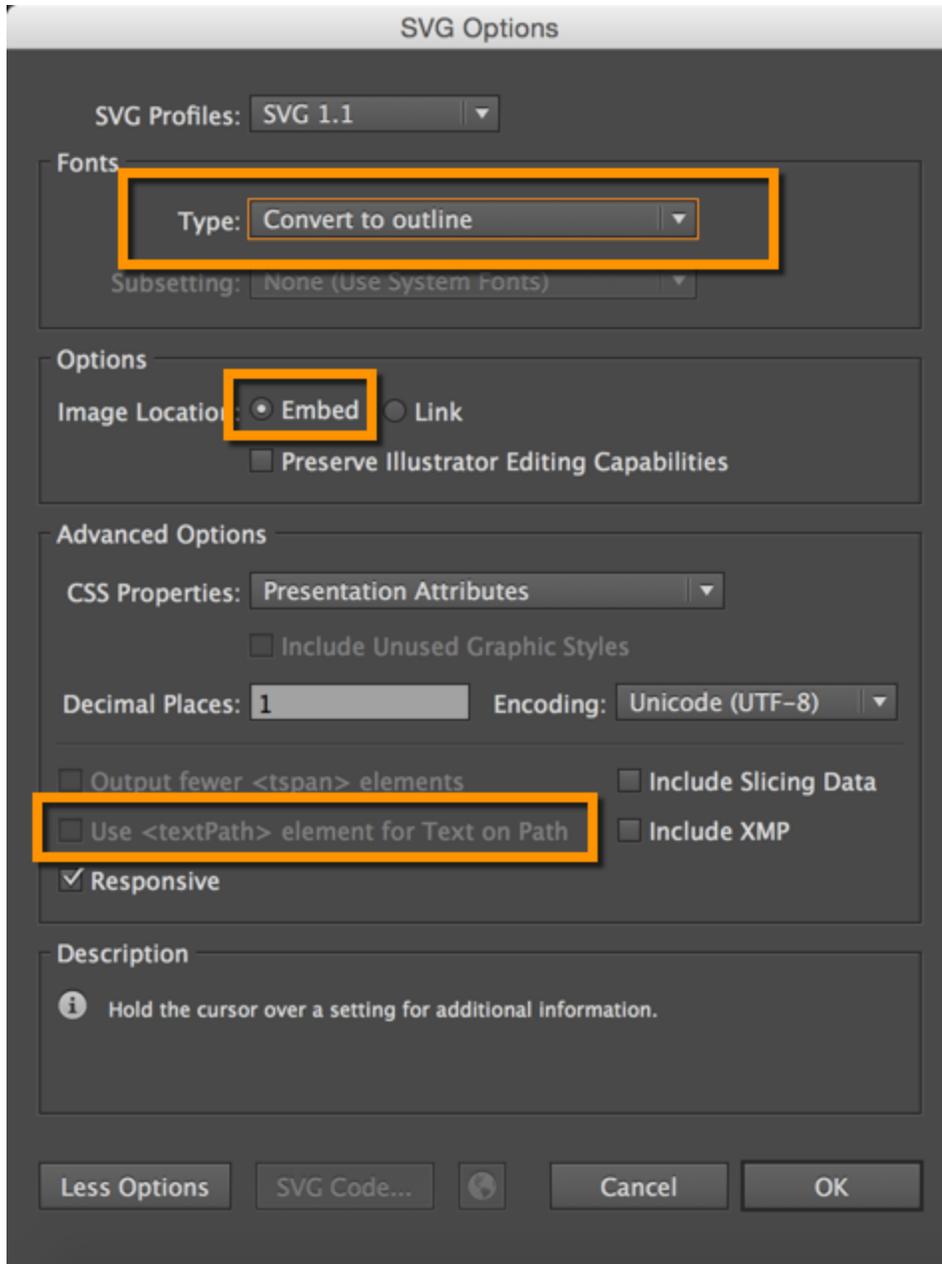
Here's the result:

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.

Labels

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

About labels

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

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

| Default | Primary | Success | Info | Warning | Danger |

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

Links

Summary: When creating links, although you can use standard HTML or Markdown, this approach is usually susceptible to a lot of errors and broken links. A better approach to handling links is to use references to a YML file.

Link strategies

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.

Create an external link

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>
```

Linking to internal pages

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.

Managed 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.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:

```
doc_getting_started:
  title: "Getting started with this theme"
  url: "doc_getting_started.html"
  link: "<a href='doc_getting_started.html'>Getting started wit
h this theme</a>"
{% raw %}
```

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

```
{% raw %}
{{site.data.urls.doc_getting_started.link}}
{% raw %}
```

This will insert the following into your topic:

```
{% raw %}
<a href='doc_getting_started.html'>Getting started with this th
eme</a>
```

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

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

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

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

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

Always make sure your TOC page is accurate

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

Checking for broken links

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

- page 0

- see .

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

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

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

Relative link paths

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 assets into subfolders.

For example, to link to a file stored in files/doc/whitepaper.pdf, you would use "files/doc/whitepaper.pdf" as the link.

Why not put pages too into subfolders? If you put a page into a subfolder, then links to the stylesheets, JavaScript, and other sources will fail. On those sub-folder pages, you'd need to use `../` to move up a level in the directory to access the stylesheets, JavaScript, etc. But if you have some pages in folders on one level, others in sub-sub-folders, and others in the root, trying to guess which files should contain `../` or `../../` or nothing at all and which shouldn't will be a nightmare.

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.

Limitations with links

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.

NavTabs

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

Common uses

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

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

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

Navtabs demo

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

Profile   About   Match

Profile

Praesent sit amet fermentum leo. Aliquam feugiat, nibh in u ltrices mattis, felis ipsum venenatis metus, vel vehicula libero mauris a enim. Sed placerat est ac lectus vestibulum tempor. Quisque ut condimentum massa. Proin venenatis leo id urna cursus blandit. Vivamus sit amet hendrerit metus.

Code

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

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

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

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

Design constraints

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

Appearance in the mini-TOC

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

Must use HTML

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

Match up ID tags

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

Set an active tab

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

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

Sets a cookie

The navtabs are part of Bootstrap, but this theme sets a cookie to remember the last tab's state. The js/customscripts.js file has a long chunk of JavaScript that sets the cookie. The JavaScript comes from this StackOverflow thread (http://stackoverflow.com/questions/10523433/how-do-i-keep-the-current-tab-active-with-twitter-bootstrap-after-a-page-reload).

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

Functionality to implement

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

Video embeds

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.

About Video JS

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.

> ❶ **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.

Tables

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

Multimarkdown Tables

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

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

This renders to the following:

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

jQuery datables

You also have the option of using a jQuery datatable (https://www.datatables.net/), which gives you some more options. If you want to use a jQuery datatable, then add `datatable: 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.

Syntax highlighting

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

About syntax highlighting

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

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

This looks as follows:

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

Fenced code blocks require a blank line before and after.

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

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

It renders the same:

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

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

```
highlighter: pygments
```

You can use another highlighter such as `rouge` .

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

Available Pygments lexers

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

Conditional logic

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

About Liquid and conditional statements

If you want to create different outputs for different audiences, you can do all of this using a combination of Jekyll's Liquid markup and values in your configuration file.

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.

Where to store filtering values

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.

Required conditional attributes

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.

Conditional logic based on config file value

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`).

Or operator

You can use more advanced Liquid markup for conditional logic, such as an `or` command. See Shopify's Liquid documentation (http://docs.shopify.com/themes/liquid-documentation/basics/operators) for more details.

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

```
{% if 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.

Unless operator

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

```
{% unless site.print == true %}
...
{% 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.

Storing conditions in the _data folder

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

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

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

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.

Specifying the location for _data

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

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

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

```
data_source: data_alpha
```

Then create a folder called _data_alpha.

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

```
data_source: data_beta
```

Then create a folder called _data_beta.

Conditional logic based on page namespace

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 %}
```

Conditions versus includes

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

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

Content reuse

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

About content reuse

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

```
{% include 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.

Page-level variables

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

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

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

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

Honestly, I haven't found a tremendous 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.

Build arguments

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.

How to build Jekyll sites

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

```
jekyll build
```

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

```
jekyll serve
```

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

```
jekyll serve --config configs/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 `../doc_writers`.

Shortcuts for the build arguments

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 doc_multibuild_web.sh file in the root directory.

My preference is to add the scripts to profiles in iTerm. See {"title"=>"iTerm profiles", "url"=>"doc_iterm_profiles.html", "link"=>"iTerm profiles (page 125)"} for more details.

Stop a server

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

You can kill the server process using these commands:

```
ps aux | grep jekyll
```

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

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

To kill all Jekyll instances, use this:

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

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

Themes

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

Theme options

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

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

Theme differences

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

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

Generating PDFs

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

PDF overview

This process for creating a PDF relies on Prince XML to transform the HTML content into PDF. Prince costs about $500 per license. That might seem like a lot, but if you're creating a PDF, you're probably working for a company that sells a product, so you likely have access to some resources.

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

It may seem like the setup is somewhat cumbersome, but it doesn't take long. Once you set it up, building a pdf is just a matter of running a couple of commands.

Also, creating a PDF this way gives you a lot more control and customization capabilities than with other methods for creating PDFs. If you know CSS, you can entirely customize the output.

Demo

You can see an example of the finished product here:

⬇ Designers PDF Download

1. Set up Prince

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

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

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

The PDF configuration file will build on the settings in the regular configuration file but will some additional fields. Here's the configuration file for the config_designers.yml file for this theme:

```
destination: ../doc_designers-pdf
url: "http://127.0.0.1:4002"
baseurl: "/doc_designers"
port: 4002
print: true
print_title: Jekyll Documentation Theme for Designers
print_subtitle: version 3.0
defaults:
  -
    scope:
      path: ""
      type: "pages"
    values:
      layout: "page_print"
      comments: true
      search: true
```

> **ⓘ Note:** Although you're creating a PDF, you must still build a 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`.

Unlike the other configuration files, the PDF configuration files require a `url` and `baseurl`. This is because the Prince utility needs to access the pages in a specific place. While you could probably set up locations via absolute paths to file folders, it's easier just to provide the locations here as `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 `print: true` 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.print` is true or not. If it's true, 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.

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

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

- titlepage.html

- tocpage.html

These pages should appear in your sidebar YML file (in this theme, sidebar_doc.yml):

```
entries:
  - title: Sidebar
    subcategories:
      - title: Frontmatter
        audience: writers, designers
        platform: all
        product: all
        version: all
        web: false
        items:
          - title: Title Page
            url: /titlepage.html
            audience: writers, designers
            platform: all
            product: all
            version: all
            web: false

          - title: Table of Contents
            url: /tocpage.html
            audience: writers, designers
            platform: all
            product: all
            version: all
            web: false
```

Leave these pages here in your sidebar. (The `web: false` property means they won't appear in your online TOC because the conditional logic of the sidebar.html checks whether `web` is equal to `false` 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.

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

4. Customize your headers and footers

Open up the css/printstyles.css file and customize what you want for the headers and footers. At the very least, customize the email address 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, 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 after links to web resources, the URL should be inserted
instead of the page number:

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

This style sets your 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";    }
}
```

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, 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: September 13, 2015";
    });
</script>

<script>
    Prince.addScriptFunc("guideName", function() {
        return "Jekyll Documentation Theme for Designers 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).

5. Customize the doc_multiserve_pdf.sh script

Open the doc_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

# serve all di print deliverables

# Writers
echo "Serving Writers PDF"
jekyll serve --detach --config configs/config_writers.yml,confi
gs/config_writers_pdf.yml

# Designers
echo "Serving Designers PDF"
jekyll serve --detach --config configs/config_designers.yml,con
figs/config_designers_pdf.yml
```

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 PDF configurations for our two
projects. This web version is where Prince will go to get its content.

6. Configure the Prince scripts

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

```
prince --javascript --input-list=../doc_designers-pdf/prince-fi
le-list.txt -o /Users/tjohnson/projects/documentation-theme-jek
yll/doc_designers_pdf.pdf;
```

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. Then when I build
the web output, the PDF is included and something I can refer to.

7. Add a download button for the PDF

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

```
<a target="_blank" class="noCrossRef" href="doc_designers_pdf.p
df"><button type="button" class="btn btn-default" aria-label="L
eft Align"><span class="glyphicon glyphicon-download-alt" ari
a-hidden="true"></span> Designers PDF Download</button></a>
```

Here's what that looks like:

[⬇ Designers PDF Download]

8. Run the scripts

To generate the PDF, you just run several scripts that have the commands
packaged up:

1. First run doc_multiserve_pdf.sh to serve up the PDF sites. The
   commands will detach the site from the preview server so that you can
   serve up multiple Jekyll sites in the same command terminal.

2. Then run doc_multibuild_pdf.sh to build the PDF files.

3. Now run doc_multibuild_web.sh to build the web version that includes the
   generated PDF files.

> ❶ **Note:** If you don't like the style of the PDFs, just adjust the styles in the
> printstyles.css file.

JavaScript conflicts

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

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

However, the PDF will still build.

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

Then surround the JavaScript with conditional tags like this:

```
{% unless site.print == true %}
javascript content here ...
{% endunless %}
```

For more detail about using `unless` in conditional logic, see Conditional logic (page 87). What this code means is basically the opposite of `if site.print == true`.

Excluding files

> **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.

About 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:
    - doc_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.

Exclude strategies

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.

Excluding draft content

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.

Limitations

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.

Help APIs and UI tooltips

Summary: You can loop through files and generate a JSON file that developers can consume like a help API. Developers can pull in values from the JSON into interface elements, styling them as popovers for user interface text, for example. The beauty of this method is that the UI text remains in the help system and isn't hard-coded into the UI.
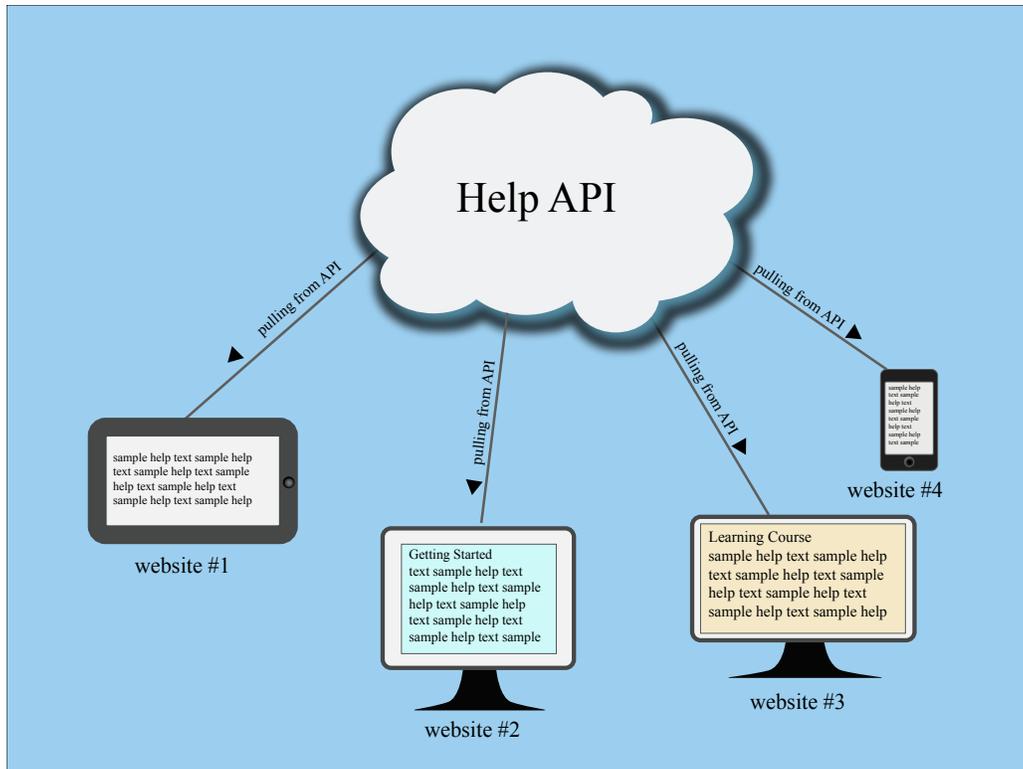
Full code demo of content API

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

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

In this demo, the popovers pull in and display content from the information in an external tooltips.json file located on a different host.

Diagram overview

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

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

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

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

1. Create a "collection" for the help content (optional)

A collection is another content type that extends Jekyll beyond the use of pages and posts. Here I'm calling the collection "tooltips." You could also just use pages, but if you have a lot of content, it will take longer to look up information in the file because the lookup will have to scan through all your site content instead of just the tooltips.

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

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

In your Jekyll project, create a new folder called "_tooltips" and put every page that you want to be part of that tooltips collection inside that folder.

2. Create pages in your collection

Create pages inside your new tooltips collection (that is, inside the _tooltips folder). Each page needs only a unique `id` in the frontmatter. Here's an example:

```
---
id: basketball
---

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

You need to create a separate page for each resource you want to deliver. In this setup, the definition of basketball is stored in a data file call definitions inside the _data folder so that we can re-use it in other parts of the help as well. (This additional re-use is covered later on this page.)

3. Create a JSON file that loops through your collection pages

Add the following to a file and call it tooltips.json:

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

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

> ☑ **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.

Store this tooltips.json file in your root directory. You can add different fields depending on how you want the JSON to be structured. Here I just have to fields: `id` and `body`. And the JSON is looking just in the tooltips collection that I created.

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.

You could 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 `getting_started: true` or something. Or you could put it into a 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.

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

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, then 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.

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

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

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

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.

5. Explain how developers can access the help

Developers can access the help using the `.get` method from jQuery, among other methods. Here's an example of how to get a page with the ID of `basketball`:

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

var url = "{url}/tooltips.json";


$.get( url, function( data ) {

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

        });
    });

});
</script>
```

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

> ❶ **Warning: Note:** Make sure your JSON file is valid. Otherwise, this method won't work. I use the JSON Formatter extension for Chrome (https://chrome.google.com/webstore/detail/json-formatter/bcjindcccaagfpapjjmafapmmgkkhgoa?hl=en). When I go to the tooltips.json page in my browser, the JSON content — if valid — is nicely formatted (and includes some color coding). If the file isn't valid, it's not formatted and there isn't any color. You can also check the JSON formatting using JSON Formatter and Validator (http://jsonformatter.curiousconcept.com/). If your JSON file isn't valid, identify the problem area using the validator and troubleshoot the file causing issues. It's usually due to some code that isn't escaping correctly.

Why `data-content`? Well, in this case, I'm using Bootstrap popovers (http://getbootstrap.com/javascript/#popovers) to display the tooltip content. The `data-content` attribute is how Bootstrap injects popovers.

Here's the section on the page where the popover is inserted:

```
<p>Basketball <span class="glyphicon glyphicon-info-sign" id="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
    });
```

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.

6. Create easy links to embed the help in your help site

You might also want to insert the same content into different parts of your help site. For example, if you have tooltips providing definitions for fields, you'll probably want to create a page in your help that lists those same definitions. You could use the same method developers use to pull help content into their applications. But it will probably be easier to simply use Jekyll's tags for doing it.

Here's how you would reuse the content:

```
<h2>Reuse Demo</h2>


<table>
<thead>
<tr>
<th>Sport</th>
<th>Comments</th>
</tr>
</thead>
<tbody>

<tr>
<td>Basketball</td>
<td>{{site.data.definitions.basketball}}</td>
</tr>

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

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

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

And here's the code:

Reuse Demo

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

Search configuration

Summary: The search feature uses JavaScript to look for keyword matches in a JSON file. The results show instant matches, but it doesn't provide a search results page like Google. Also, sometimes invalid formatting can break the JSON file.

About search

The search is configured through the search.json file in the root directory. Take a look at that code if you want to change what fields are included.

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

However, the search doesn't work like google — you can't hit return and see a list of results on the search results page, with the keywords in bold. Instead, this search shows a list of page titles that contain keyword matches. It's fast, but simple.

Excluding pages form search

By default, every page is included in the search. Depending on the type of content you're including, you may find that some pages will break the JSON formatting. If that happens, then the search will no longer work.

If you want to exclude a page from search add `search: exclude` in the frontmatter.

Troubleshooting search

You should exclude any files from search that you don't want appearing in the search results. For example, if you have a tooltips.json file or prince-file-list.txt, don't include it, as the formatting will break the JSON format.

If any formatting in the search.json file is invalid (in the build), search won't work. You'll know that search isn't working if no results appear when you start typing in the search box.

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

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

```
    "body": "{{ page.content | strip_html | strip_newlines |
replace: '\', '\\\\' | replace: '"', '\\"' }}",
```

However, 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.)

Customizing search results

At some point, you may want to customize the search results more. Here's a little more detail that will be helpful. The search.json file retrieves various page values:

```
    {% if page.search == true %}
      {
      "title": "{{ page.title | escape }}",
      "tags": "{{ page.tags }}",
      "keywords": "{{page.keywords}}",
      "url": "{{ page.url | replace: "/", "" }}",
      "last_updated": "{{ page.last_updated }}",
      "summary": "{{page.summary}}",
      "body": "{{ page.content | strip_html | strip_newlines |
replace: '\', '\\\\' | replace: '"', '\\"' }}"
      }
```

The _includes/topnav.html file then makes use of these values:

```
<!-- start search -->
<div id="search-demo-container">
<input type="text" id="search-input" placeholder="search...">
<ul id="results-container"></ul>
</div>
<script src="js/jekyll-search.js" type="text/javascript"></script>
<script type="text/javascript">
SimpleJekyllSearch.init({
searchInput: document.getElementById('search-input'),
resultsContainer: document.getElementById('results-container'),
dataSource: 'search.json',
searchResultTemplate: '<li><a href="{url}" title="Search config
uration">{title}</a></li>',
noResultsText: 'No results found.',
limit: 10,
fuzzy: true,
})
</script>
<!-- end search -->
</li>
```

Where you see `{url}` and `{title}`, the search is retrieving the values for these as specified in the search.json file.

At some point, you may want to add in the `{summary}` as well. You could create a dedicated search page that could include the summary as an instant result as you type.

iTerm profiles

> **Summary:** Set up profiles in iTerm to facilitate the build process with just a few clicks. This can make it a lot easier to quickly build multiple outputs.

About iTerm profiles

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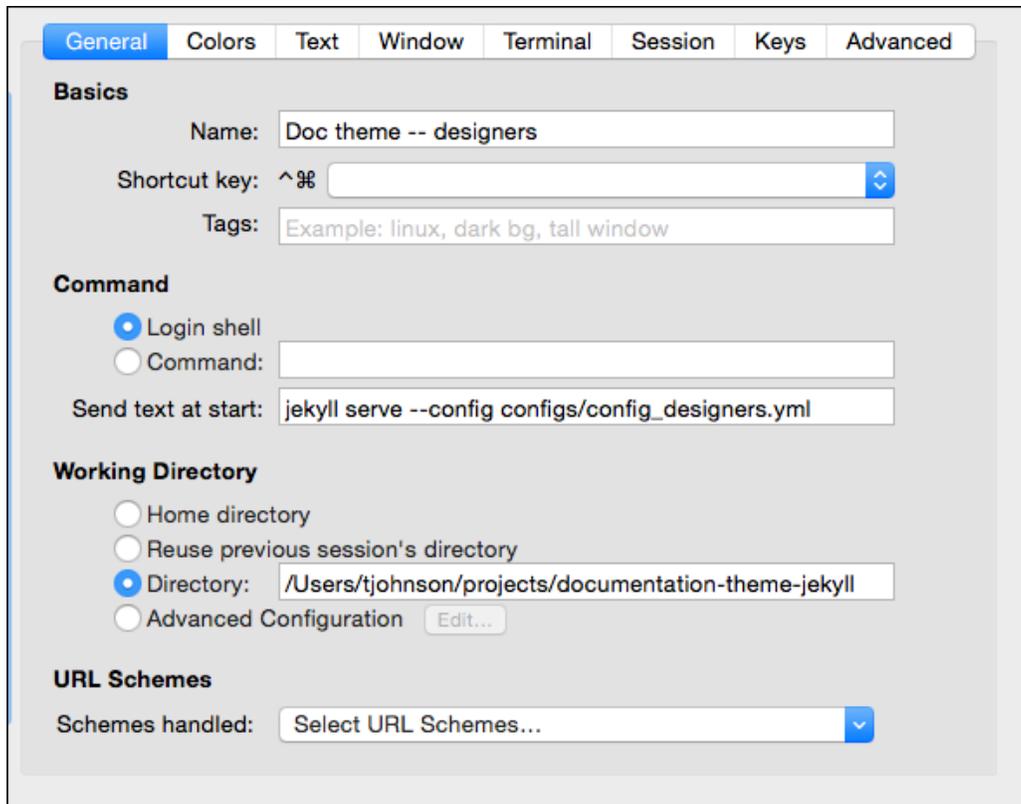
Set up profiles

1. Open iTerm and go to **Profiles > Open Profiles.**

2. Click **Edit Profiles**.

3. Click the + button in the lower-left corner to create a new profile.

4. In the **Name** field, type a name describing the output, such as `Doc theme -- designers`.

5. In the **Send text at start** field, type the command for the build script, such as this:

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

   Leave the Login shell option selected.

6. In the Working Directory section, select **Directory** and enter the directory for your project, such as **/Users/tjohnson/projects/documentation-theme-jekyll**.

7. Close the profiles panel.

Here's an example:

Launching a profile

1. In iTerm, make sure the Toolbar is shown. Go to **View > Toggle Toolbar**.

2. Click the **New** button and select your profile.

> ☑ **Tip:** When you're done with the session, make sure to click **Ctrl+C**.

Pushing builds to server

Summary: You can push your build to AWS using commands from the command line. By including your copy commands in commands, you can package all of the build and deploy process into executable scripts.

Pushing to AWS S3

If you have the AWS Command Line Interface installed and are pushing your builds to AWS, the following commands show how you can build and push to an AWS location from the command line:

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

#aws s3 cp ~/users/tjohnson/projects/documentation-theme-jekyl
l-builds/doc_designers s3://[aws path]/documentation-theme-jeky
ll/doc_designers --recursive
```

The first path is the local location; the second path is the destination.
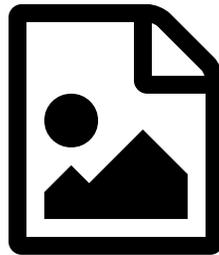
Pushing to a regular server

If you're pushing to a regular server that you can ssh into, you can use `scp` commands to push your build. Here's an example:

```
scp -r /users/tjohnson/projects/documentation-theme-jekyll-buil
ds/doc_writers name@domain:/var/www/html/documentation-theme-je
kyll/doc_writers
```

Similar to the above, the first path is the local location; the second path is the destination.
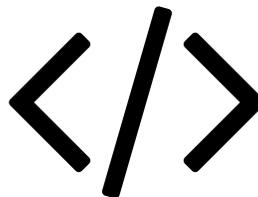
Knowledge-base layout

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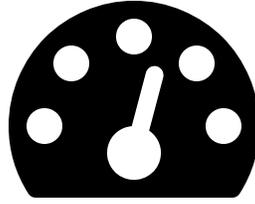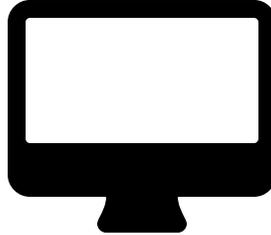
Getting Started

Navigation

Single-sourcing

Publishing

Special layouts

Formatting

Generating a list of all pages with a certain tag

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

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

Getting started pages:

- About this theme (page 0)

- Customizing the theme (page 16)

- Getting started with this theme (page 3)

- Introduction (page 1)

- Pages (page 25)

- Support (page 0)

- Supported features (page 19)

- Troubleshooting (page 0)

- WebStorm Text editor (page 32)

Scroll layout

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 133): "red fruit at the store",
  "bananas" (page 133): "yellow bananas in a bunch",
  "carrots" (page 133): "orange vegetables that grow in th
e ground",
  "dingbats" (page 133): "a type of character symbol on a c
omputer",
  "eggs" (page 133): "chickens lay them, and people eat the
m",
  "falafel" (page 134): "a Mediterranean sandwich consistin
g of lots of different stuff i don't know much about",
  "giraffe" (page 134): "tall animal, has purple tongue",
  "hippo" (page 134): "surprisingly dangerous amphibian",
  "igloo" (page 134): "an ice shelter made by eskimos",
  "jeep (page 134): "the only car that starts with a j",
  "kilt" (page 134): "something worn by scottish people, no
t a dress",
  "lamp" (page 134): "you use it to read by your bedside a
t night"
  "manifold" (page 134): "an intake mechanism on a car, lik
e a valve, i think",
  "octopus" (page 134): "eight tentacles, shoots ink, live
s in dark caves, very mysterious",
  "paranoia" (page 135): "the constant feeling that others
are out to get you, conspiring against your success",
  "qui" (page 135): "a life force that runs through your bo
dy",
  "radical" (page 135): "someone who opposes the status qu
o in major ways",
  "silly" (page 135): "how I feel writing this dummy copy",
  "taffy" (page 135): "the sweets children like the most an
d dentists hate the worst",
  "umbrella" (page 135): "an invention that has not had an
y advancements in 200 years",
  "vampire" (page 135): "a paranormal figure that is surpri
singly in vogue despite its basic nature",
  "washington" (page 135): "the place where tom was born",
  "xylophone" (page 136): "some kind of pinging instrument
used to sound chime-like notes",
  "yahoo" (page 136): "an expression of exuberance, said un
der breath when something works right",
  "zeta" (page 136): "the way british people pronounce z",
  "alpha" (page 136): "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 136): "the period of time when something is
finished but undergoing testing by a group of people.",
  "cappa" (page 136): "how italians refer to their basebal
l caps",
  "dunno" (page 136): "informal expression for 'don't kno
w'"
  }
```

apples

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.

bananas

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.

carrots

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.

dingbats

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

eggs

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.

falafel

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,

giraffe

Nullam venenatis at lectus sed pharetra. Sed hendrerit ligula lectus, non pellentesque diam faucibus sit amet. Aliquam dictum hendrerit pellentesque. Cras eu nisl sagittis, faucibus velit sit amet, sagittis odio. Donec vulputate ex vitae purus

hippo

Cras nec pretium nulla. Suspendisse tempus tortor vel venenatis pulvinar. Integer varius tempor enim fringilla tincidunt. Phasellus magna turpis, auctor vitae elit eget, fringilla pellentesque est. Phasellus ut porta risus. Curabitur iaculis sapien sed venenatis auctor. Integer eu orci at lectus eleifend auctor id rutrum urna.

Fusce rhoncus elit sed quam laoreet placerat. Praesent lacinia metus quis felis mollis, ac facilisis risus consequat. Phasellus laoreet feugiat lacus. Etiam a neque est.

jeep

Nulla vitae metus rutrum, condimentum orci nec, maximus est. Aenean sit amet ante nec elit dignissim faucibus eget quis quam.

kilt

Morbi maximus, erat vel rhoncus sagittis, dolor purus dignissim ante, sit amet pharetra ex justo vitae ipsum. Nulla consequat interdum neque

lamp

Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Mauris aliquam dapibus blandit. Donec porta, enim hendrerit venenatis vulputate, orci diam lacinia nibh, faucibus rutrum dolor dui ut quam.

manifold

Donec finibus massa vel nisi ullamcorper, vitae ornare enim euismod. Aliquam auctor quam erat. Duis interdum rutrum orci, ac interdum urna pharetra eget.

octopus

Nulla id egestas enim. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse potenti. Curabitur eu lobortis ligula.

paranoia

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.

qui

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

radical

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.

silly

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.

taffy

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.

umbrella

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.

vampire

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.

washington

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.

xylophone

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.

yahoo

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.

Nullam ac suscipit odio. Curabitur viverra arcu ut egestas sollicitudin. Fusce sodales varius lectus ut tristique. Etiam eget nunc ornare, aliquet tortor eget, consequat mauris. Integer sit amet fermentum augue.

alpha

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.

Mauris a libero vel enim pharetra interdum non a quam. Sed tincidunt ut elit sed dignissim. Suspendisse vitae tellus dapibus, fermentum lacus ac, fermentum lacus. Nam ante odio, fringilla ac elementum a, mollis sed tellus.

cappa

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

dunno

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.

> ❶ **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.

Shuffle layout

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

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

All	Getting Started	Formatting	Publishing	Content types

Single Sourcing	Special Layouts

Getting started

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

- About this theme (page 0)
- Customizing the theme (page 16)
- Getting started with this theme (page 3)
- Pages (page 25)
- Support (page 0)
- Supported features (page 19)
- Troubleshooting (page 0)

Content types

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

- Collections (page 38)
- Generating PDFs (page 98)
- Help APIs and UI tooltips (page 112)
- Pages (page 25)
- Series pages (page 35)

Formatting

- WebStorm Text editor (page 32)

- Introduction (page 1)

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

- Tooltips (page 55)

- Alerts (page 56)

- Glossary layout (page 141)

- Links (page 70)

- Icons (page 60)

- Images (page 66)

- Labels (page 69)

- NavTabs (page 75)

- Pages (page 25)

- Syntax highlighting (page 85)

- Tables (page 81)

- Video embeds (page 78)

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 87)

- Setting configuration options (page 7)

- Content reuse (page 92)

Publishing

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

- Build arguments (page 94)

- Setting configuration options (page 7)

- Generating PDFs (page 98)

Special Layouts

These pages highlight special layouts outside of the conventional page and TOC hierarchy.

> ❶ **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.

FAQ layout

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

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

Glossary layout

> **Summary:** Your glossary page can take advantage of definitions stored in a data file. This gives you the ability to reuse the same definition in multiple places. Additionally, you can use Bootstrap classes to arrange your definition list horizontally.

You can create a glossary for your content. First create your glossary items in a data file such as glossary.yml.

Then create a page and use definition list formatting, like this:

```
<dl class="dl">

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

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

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

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

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

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

</dl>
```

Here's what that looks like:

fractious

Like a little mischevious child, full of annoying and constant trouble.

gratuitous

Something that is unwarranted and uncouth, like the social equivalent of a flagrant foul.

haughty

Proud and flaunting it. Holding your head high up like a snooty, too-good-for-everything rich person.

gratuitous

Something that is unwarranted and uncouth, like the social equivalent of a flagrant foul.

impertinent

Someone acting rude and insensitive to others.

intrepid

Brave and courageous especially in a difficult, dangerous situation.

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

Horizontally styled definiton lists

You can also change the definition list (`dl`) class to `dl-horizontal` . This is a Bootstrap specific class. If you do, the styling looks like this:

fractious

Like a little mischevious child, full of annoying and constant trouble.

gratuitous

Something that is unwarranted and uncouth, like the social equivalent of a flagrant foul.

haughty

Proud and flaunting it. Holding your head high up like a snooty, too-good-for-everything rich person.

gratuitous

Something that is unwarranted and uncouth, like the social equivalent of a flagrant foul.

impertinent

Someone acting rude and insensitive to others.

intrepid

Brave and courageous especially in a difficult, dangerous situation.

If you squish your screen small enough, at a certain breakpoint this style reverts to the regular `dl` class.

Although I like the side-by-side view for shorter definitions, I found it problematic with longer definitions.

Tag archives overview

Summary: This is an overview to the tag archives section. Really the only reason this section is listed explicitly in the TOC here is to demonstrate how to add a third-level to the navigation.

Reasons for tags

Tags provide alternate groupings for your content. In the documentation for this theme, there are a number of equally plausible ways I could have grouped the content. The folder names and items I chose for each item could have been grouped in other ways with good reason.

Tags allow you to go beyond the traditional hierarchical classification and provide other groupings. For example, the same item can belong to two different groups. You can also introduce other dimensions not used in your table of contents, such as platform-specific tags or audience-specific tags.