성신여대 오픈소스 툴 실습

Summary: 실습 방법.

Build the Theme

Sidebar syntax

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

Each `folder` or `subfolder` must contain a `title` and `output` property. Each `folderitem` or `subfolderitem` must contain a `title`, `url`, and `output` property.

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

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

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

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

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

For more detail on the sidebar, see Sidebar navigation (page 61) and YAML tutorial (page 65).

Relative links and offline viewing

Page frontmatter

When you write pages, include these same frontmatter properties with each page:

```
---
title: "Some title"
tags: [sample1, sample2]
keywords: keyword1, keyword2, keyword3
last_updated: Month day, year
summary: "optional summary here"
sidebar: sidebarname
permalink: filename.html
---
```

(You will customize the values for each of these properties, of course.)

For titles, surrounding the title in quotes is optional, but if you have a colon in the title, you must surround the title with quotation marks. If you have a quotation mark inside the title, escape it first with a backlash `\`.

Values for `keywords` get populated into the metadata of the page for SEO.

Values for `tags` must be defined in your _data/tags.yml list. You also need a corresponding tag file inside the tags folder that follows the same pattern as the other tag files shown in the tags folder. (Jekyll won't auto-create these tag files.)

If you don't want the mini-TOC to show on a page (such as for the homepage or landing pages), add `toc: false` in the frontmatter.

The `permalink` value should be the same as your filename and include the ".html" file extension.

For more detail, see Pages (page 33).

Where to store your documentation topics

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

Note that product1, product2, and mydoc are all just sample content to demonstrate how to add multiple products into the theme. You can freely delete that content.

For more information, see Pages (page 33) and Posts (page 40).

Configure the top navigation

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

For external URLs, use `external_url` in the item property, as shown in the example topnav.yml file. For internal links, use `url` the same was you do in the sidebar data files.

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

Generating PDF

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

If you're on Windows, install Git Bash client (https://git-for-windows.github.io/) rather than using the default Windows command prompt.

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

See the section on [Generating PDFs][mydoc_generating_pdfs] for more details about setting the theme up for this output.

Blogs / News

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

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

See Posts (page 40) for more information.

Markdown

This theme uses kramdown markdown (http://kramdown.gettalong.org/). kramdown is similar to Github-flavored Markdown, except that when you have text that intercepts list items, the spacing of the intercepting text must align with the spacing of the first character after the space of a numbered list item. Basically, with your list item numbering, use two spaces after the dot in the number, like this:

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

When you want to insert paragraphs, notes, code snippets, or other matter in between the list items, use four spaces to indent. The four spaces will line up with the first letter of the list item (the **F**irst or **S**econd or **T**hird).

```
1.  First item

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

2. Second item

    Some pig!

3. Third item
```

See the topics under “Formatting” in the sidebar for more information.

Automated links

If you want to use an automated system for managing links, see Automated Links (page 0). This approach automatically creates a list of Markdown references to simplify linking.

Other instructions

The content here is just a getting started guide only. For other details in working with the theme, see the various sections in the sidebar.

Introduction

Overview

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

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

Survey of features

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

- Bootstrap framework
- Navgoco multi-level sidebar (http://www.komposta.net/article/navgoco) for table of contents
- Ability to specify different sidebars for different products
- Top navigation bar with drop-down menus
- Notes, tips, and warning information notes
- Tags for alternative navigation
- Advanced landing page layouts from the Modern Business theme (http://startbootstrap.com/template-overviews/modern-business/)
  .

Getting started

To get started, see Getting Started (page 1).

Supported features

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

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

Supported features

Features	Supported	Notes
Content re-use	Yes	Supports re-use through Liquid. You can re-use variables, snippets of code, entire pages, and more. In DITA speak, this includes conref and keyref.
Markdown	Yes	You can author content using Markdown syntax. This is a wiki-like syntax for HTML that you can probably pick up in 10 minutes. Where Markdown falls short, you can use HTML. Where HTML falls short, you use Liquid, which is a scripting that allows you to incorporate more advanced logic.
Responsive design	Yes	Uses Bootstrap framework for responsive design.

Features	Supported	Notes
Translation	Yes	I haven’t done a translation project yet (just a pilot test). Here’s the basic approach: Export the HTML pages and send them to a translation agency. Then create a new project for that language and insert the translated pages. Everything will be translated.
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
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	Yes	There is a simple blogging feature. This appears as "news" and is intended to promote news that applies across products.
Versioning	Yes	Jekyll doesn't version your files. You upload your files to a version control system such as Github. Your files are versioned there.
PC platform	Yes	Jekyll runs on Windows. Although the experience working on the command line is better on a Mac, Windows also works, especially now that Jekyll 3.0 dropped dependencies on Python, which wasn't available by default on Windows.

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
Themes	Yes	You can have different themes for different outputs. If you know CSS, theming both the web and print outputs is pretty easy.
Open source	Yes	This theme is entirely open source. Every piece of code is open, viewable, and editable. Note that this openness comes at a price — it's easy to make changes that break the theme or otherwise cause errors.
Offline viewing	Yes	This theme uses relative linking throughout, so you can view the content offline and on any webserver without configuring urls and baseurls in your configuration file.

Features not available

The following features are not available.

Features	Supported	Notes
CMS interface	No	Unlike with WordPress, you don't log into an interface and navigate to your files. You work with text files and preview the site dynamically in your browser. Don't worry – this is part of the simplicy that makes Jekyll awesome. I recommend using WebStorm as your text editor.
WYSIWYG interface	No	I use WebStorm to author content, because I like working in text file formats. But you can use any Markdown editor you want (e.g., Lightpaper for Mac, Marked) to author your content.

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

About the theme author

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

My name is Tom Johnson, and I'm a technical writer, blogger, and podcaster based in San Jose, California. My blog is here: http://idratherbewriting.com (http://idratherbewriting.com). I write several posts there a week. See my blog's about page (http://idratherbewriting.com/aboutme/) for more details about me.

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

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

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

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

Support

Summary: Contact me for any support issues.

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

About Ruby, Gems, Bundler, and other prerequisites

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

About Ruby

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

About Ruby Gems

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

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

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

Rubygem package managers

Bundler (http://bundler.io/) is a gem package manager for Ruby, which means it goes out and gets all the gems you need for your Ruby programs. If you tell Bundler you need the jekyll gem (https://rubygems.org/gems/jekyll), it will retrieve all the dependencies on the jekyll gem as well – automatically.

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

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

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

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

> Bundler provides a consistent environment for Ruby projects by tracking and installing the exact gems and versions that are needed.
>
> Bundler is an exit from dependency hell, and ensures that the gems you need are present in development, staging, and production. Starting work on a project is as simple as bundle install.

Gemfiles

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

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

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

The source indicates the site where Bundler will retrieve the gems: https://rubygems.org (https://rubygems.org).

The gems it retrieves are listed separately on each line.

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

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

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

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

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

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

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

If it adds another digit, the scope is affected:

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

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

If it looks like this:

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

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

See this Stack Overflow post (http://stackoverflow.com/questions/5170547/what-does-tilde-greater-than-mean-in-ruby-gem-dependencies) for more details.

Gemfile.lock

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

```
GEM
  remote: https://rubygems.org/
  specs:
    RedCloth (4.2.9)
    activesupport (4.2.5.1)
      i18n (~> 0.7)
      json (~> 1.7, >= 1.7.7)
      minitest (~> 5.1)
      thread_safe (~> 0.3, >= 0.3.4)
      tzinfo (~> 1.1)
    addressable (2.3.8)
    coffee-script (2.4.1)
      coffee-script-source
      execjs
    coffee-script-source (1.10.0)
    colorator (0.1)
    ethon (0.8.1)
      ffi (>= 1.3.0)
    execjs (2.6.0)
    faraday (0.9.2)
      multipart-post (>= 1.2, < 3)
    ffi (1.9.10)
    gemoji (2.1.0)
    github-pages (52)
      RedCloth (= 4.2.9)
      github-pages-health-check (= 1.0.1)
      jekyll (= 3.0.3)
      jekyll-coffeescript (= 1.0.1)
      jekyll-feed (= 0.4.0)
      jekyll-gist (= 1.4.0)
      jekyll-mentions (= 1.0.1)
      jekyll-paginate (= 1.1.0)
      jekyll-redirect-from (= 0.9.1)
      jekyll-sass-converter (= 1.3.0)
      jekyll-seo-tag (= 1.3.1)
      jekyll-sitemap (= 0.10.0)
      jekyll-textile-converter (= 0.1.0)
      jemoji (= 0.5.1)
      kramdown (= 1.9.0)
      liquid (= 3.0.6)
      mercenary (~> 0.3)
      rdiscount (= 2.1.8)
      redcarpet (= 3.3.3)
      rouge (= 1.10.1)
      terminal-table (~> 1.4)
    github-pages-health-check (1.0.1)
      addressable (~> 2.3)
```

```
    net-dns (~> 0.8)
    octokit (~> 4.0)
    public_suffix (~> 1.4)
    typhoeus (~> 0.7)
  html-pipeline (2.3.0)
    activesupport (>= 2, < 5)
    nokogiri (>= 1.4)
  i18n (0.7.0)
  jekyll (3.0.3)
    colorator (~> 0.1)
    jekyll-sass-converter (~> 1.0)
    jekyll-watch (~> 1.1)
    kramdown (~> 1.3)
    liquid (~> 3.0)
    mercenary (~> 0.3.3)
    rouge (~> 1.7)
    safe_yaml (~> 1.0)
  jekyll-coffeescript (1.0.1)
    coffee-script (~> 2.2)
  jekyll-feed (0.4.0)
  jekyll-gist (1.4.0)
    octokit (~> 4.2)
  jekyll-mentions (1.0.1)
    html-pipeline (~> 2.3)
    jekyll (~> 3.0)
  jekyll-paginate (1.1.0)
  jekyll-redirect-from (0.9.1)
    jekyll (>= 2.0)
  jekyll-sass-converter (1.3.0)
    sass (~> 3.2)
  jekyll-seo-tag (1.3.1)
    jekyll (~> 3.0)
  jekyll-sitemap (0.10.0)
  jekyll-textile-converter (0.1.0)
    RedCloth (~> 4.0)
  jekyll-watch (1.3.1)
    listen (~> 3.0)
  jemoji (0.5.1)
    gemoji (~> 2.0)
    html-pipeline (~> 2.2)
    jekyll (>= 2.0)
  json (1.8.3)
  kramdown (1.9.0)
  liquid (3.0.6)
  listen (3.0.6)
    rb-fsevent (>= 0.9.3)
    rb-inotify (>= 0.9.7)
```

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

PLATFORMS
  ruby

DEPENDENCIES
  github-pages
  jekyll

BUNDLED WITH
   1.11.2
```

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

To learn more about Bundler, see Bundler's Purpose and Rationale (http://bundler.io/rationale.html).

Install Jekyll on Mac

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

Ruby and RubyGems

Ruby and RubyGems (https://rubygems.org/pages/download) are usually installed by default on Macs. Open your Terminal and type `which ruby` and `which gem` to confirm that you have Ruby and Rubygems. You should get a response indicating the location of Ruby and Rubygems.

If you get responses that look like this:

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

and

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

Great! Skip down to the Bundler (page 25) section.

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

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

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

Install Homebrew

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

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

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

```
brew update
```

Install Ruby through Homebrew

Now use Homebrew to install Ruby:

```
brew install ruby
```

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

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

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

And this:

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

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

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

Install the Jekyll gem

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

Now use `gem` to install Jekyll:

```
gem install jekyll
```

You can now use Jekyll to create new Jekyll sites following the quick-start instructions on Jekyllrb.com (http://jekyllrb.com).

Installing dependencies through Bundler

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

Bundler (http://bundler.io/) is a package manager for RubyGems. You can use it to get all the gems (or Ruby plugins) that you need for your Jekyll project.

You install Bundler by using the gem command with RubyGems:

```
gem install bundler
```

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

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

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

Serve the Jekyll Documentation theme

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

Resolve "No Github API authentication" errors

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

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

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

Note: These instructions apply to Mac OS X, but they're highly similar to Windows. These instructions are adapted from a post on Knight Codes

> (http://knightcodes.com/miscellaneous/2016/09/13/fix-github-metadata-error.html)
> . If you're on Windows, see the Knight Codes post for details instead of following along below.

To resolve the "No Github API authentication" error:

1. Follow Github's instructions to create a personal access token (https://help.github.com/articles/creating-an-access-token-for-command-line-use/)
   .
2. Open the **.bash_profile** file in your user directory:

   ```
   open ~/.bash_profile
   ```

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

   ```
   export JEKYLL_GITHUB_TOKEN=abc123abc123abc123abc123abc123abc123abc123abc123
   ```

   Replace `abc123...` with your own token that you generated in step 1.
4. Go to **[https://curl.haxx.se/ca/cacert.pem][https://curl.haxx.se/ca/cacert.pem]. Right-click the page, select **Save as**, and save the file on your computer (save it somewhere safe, where you won't delete it). Name the file **cacert**.
5. Open your **.bash_profile** file again and add this line, replacing `Users/johndoe/projects/` with the path to your cacert.pem file:

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

6. Close and restart your terminal.

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

Install Jekyll on Windows

Tip: For a better terminal emulator on Windows, use Git Bash (https://git-for-windows.github.io/). Git Bash gives you Linux-like control on Windows.

Install Ruby

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

1. Go to RubyInstaller for Windows (http://rubyinstaller.org/downloads/).
2. Under **RubyInstallers**, download and install one of the Ruby installers (usually one of the first two options).
3. Double-click the downloaded file and proceed through the wizard to install it.

Install Ruby Development Kit

Some extensions Jekyll uses require you to natively build the code using the Ruby Development Kit.

1. Go to RubyInstaller for Windows (http://rubyinstaller.org/downloads/).
2. Under the **Development Kit** section near the bottom, download one of the **For use with Ruby 2.0 and above…** options (either the 32-bit or 64-bit version).
3. Move your downloaded file onto your **C** drive in a folder called something like **RubyDevKit**.
4. Extract the compressed folder's contents into the folder.
5. Browse to the **RubyDevKit** location on your C drive using your Command Line Prompt.

   To see the contents of your current directory, type `dir` . To move into a directory, type `cd foldername` , where "foldername" is the name of the folder you want to enter. To move up a directory,

type `cd ../` one or more times depending on how many levels you want to move up. To move into your user's directory, type `/users`. The `/` at the beginning of the path automatically starts you at the root.

6. Type `ruby dk.rb init`
7. Type `ruby dk.rb install`

If you get stuck, see the official instructions for installing Ruby Dev Kit (https://github.com/oneclick/rubyinstaller/wiki/Development-Kit).

Install the Jekyll gem

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

Now use `gem` to install Jekyll:

```
gem install jekyll
```

You can now use Jekyll to create new Jekyll sites following the quick-start instructions on Jekyllrb.com (http://jekyllrb.com).

Installing dependencies through Bundler

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

Bundler (http://bundler.io/) is a package manager for RubyGems. You can use it to get all the gems (or Ruby plugins) that you need for your Jekyll project.

You install Bundler by using the gem command with RubyGems:

Install Bundler

1. Install Bundler: `gem install bundler`
2. Initialize Bundler: `bundle init`

   This will create a new Gemfile.
3. Open the Gemfile in a text editor.

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

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

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

gem 'wdm'
gem 'jekyll'
```

The wdm gem (https://rubygems.org/gems/wdm/versions/0.1.1) allows for the polling of the directory and rebuilding of the Jekyll site when you make changes. This gem is needed for Windows users, not Mac users.

5. Save and close the file.
6. Type `bundle install`.

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

Git Clients for Windows

Although you can use the default command prompt with Windows, it's recommended that you use Git Bash (https://git-for-windows.github.io/) instead. The Git Bash client will allow you to run shell scripts and execute other Unix commands.

Serve the Jekyll Documentation theme

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

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

Resolving Github Metadata errors

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

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

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

See this post on Knight Codes (http://knightcodes.com/miscellaneous/2016/09/13/fix-github-metadata-error.html)
for instructions on how to fix the error. You basically generate a personal token on Github and set it as a system variable. You also download a certification file and set it as a system variable. This resolves the issue.

Pages

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

Where to author content

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

Where to save pages

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

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

Frontmatter

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

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

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

Note that you cannot use variables in frontmatter.

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

Frontmatter	Required?	Description
title	Required	The title for the page
tags	Optional	Tags for the page. Make all tags single words, with underscores needed (rather than spaces). Separate them with commas. Enc the whole list within brackets. Also, note that tags must be add _data/tags_doc.yml to be allowed entrance into the page. This vents tags from becoming somewhat random and unstructured must create a tag page for each one of your tags following the shown in the tags folder. (Tag pages aren’t automatically create
keywords	Optional	Synonyms and other keywords for the page. This information g stuffed into the page’s metadata to increase SEO. The user wo the keywords, but if you search for one of the keywords, it will picked up by the search engine.
last_updated	Optional	The date the page was last updated. This information could hel readers trying to evaluate how current and authoritative inform is. If included, the last_updated date appears in the footer of th in rather small font.

Frontmatter	Required?	Description
summary	Optional	A 1-2 word sentence summarizing the content on the page. Thi formatted into the summary section in the page layout. Adding maries is a key way to make your content more scannable by u (check out Jakob Nielsen's site (http://www.nngroup.com/articles/corporate-blogs-front-page- ture/) for a great example of page summaries.) The only drawback wi maries is that you can't use variables in them.
permalink	Required	The permalink *must* match the filename in order for automated to work. Additionally, you must include the ".html" in the filena not put forward slashes around the permalink (this makes Jekyl the file inside a folder in the output). When Jekyll builds the site put the page into the root directory rather than leaving it in a s rectory or putting it inside a folder and naming the file index.ht Having all files flattened in the root directory is essential for rel linking to work and for all paths to JS and CSS files to be valid.
datatable	Optional	'true'. If you add `datatable: true` in the frontmatter, scripts fo jQuery Datatables plugin (https://www.datatables.net/) get inclu the page. You can see the scripts that conditionally appear by l in the _layouts/default.html page.
toc	Optional	If you specify `toc: false` in the frontmatter, the page won't ha table of contents that appears below the title. The toc refers to of jump links below the page title, not the sidebar navigation. Y probably want to hide the TOC on the homepage and product la pages.

Colons in page titles

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

Page names and excluding files from outputs

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

```
exclude:

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

Wildcards will exclude every match after the `*`.

Saving pages as drafts

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

Markdown or HTML format

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

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

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

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

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

Page names

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

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

Kramdown Markdown

Kramdown is the Markdown flavor used in the theme. This mostly aligns with Github-flavored Markdown, but with some differences in the indentation allowed within lists. Basically, Kramdown requires you to line up the indent between list items with the first starting character after the space in your list item numbering. See this blog post on Kramdown and Rouge (http://idratherbewriting.com/2016/02/21/bug-with-kramdown-and-rouge-with-github-pages/) for more details.

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

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

Automatic mini-TOCs

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

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

Headings

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

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

Result:

Second-level heading

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

Result:

Third-level heading

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

Result:

Fourth-level heading

Headings with ID Tags

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

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

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

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

Result:

Some link (page 38)

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

Specify a particular page layout

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

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

Comments

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

Posts

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

hello there.

About posts

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

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

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

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

Allowed frontmatter

The frontmatter you can use with posts is as follows:

---

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

Frontmatter	Required?	Description
title	Required	The title for the page

Frontmatter	Required?	Description
tags	Optional	Tags for the page. Make all tags single words, with underscores i needed. Separate them with commas. Enclose the whole list wit brackets. Also, note that tags must be added to _data/tags_doc.y be allowed entrance into the page. This prevents tags from becc somewhat random and unstructured. You must create a tag pag each one of your tags following the sample pattern in the tabs fo (Tag pages aren't automatically created.)
keywords	Optional	Synonyms and other keywords for the page. This information ge stuffed into the page's metadata to increase SEO. The user won the keywords, but if you search for one of the keywords, it will b picked up by the search engine.
summary	Optional	A 1-2 word sentence summarizing the content on the page. This formatted into the summary section in the page layout. Adding s maries is a key way to make your content more scannable by us (check out Jakob Nielsen's site (http://www.nngroup.com/articles/corporate-blogs-front-page-s ture/) for a great example of page summaries.) The only drawback wit maries is that you can't use variables in them.
permalink	Required	This theme uses permalinks to facilitate the linking. You specify permalink want for the page, and the _site output will put the pa to the root directory when you publish. Follow the same convent here as you do with page permalinks – list the file name followed the .html extension.

Lists

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

Bulleted Lists

This is a bulleted list:

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

Result:

- first item
- second item
- third item

Numbered list

This is a simple numbered list:

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

Result:

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

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

Complex Lists

Here’s a more complex list:

```
1.  Sample first item.

    * sub-bullet one
    * sub-bullet two

2.  Continuing the list

    1. sub-list numbered one
    2. sub-list numbered two

3.  Another list item.
```

Result:

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

Another Complex List

Here’s a list with some intercepting text:

```
1.  Sample first item.

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

2.  Continuing the list

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


    Here's a list in here:

    * first item
    * second item

3.  Another list item.

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

4. Another item.
```

Result:

1. Sample first item.

   This is a result statement that talks about something….

2. Continuing the list

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

   Here’s a list in here:

   - first item
   - second item

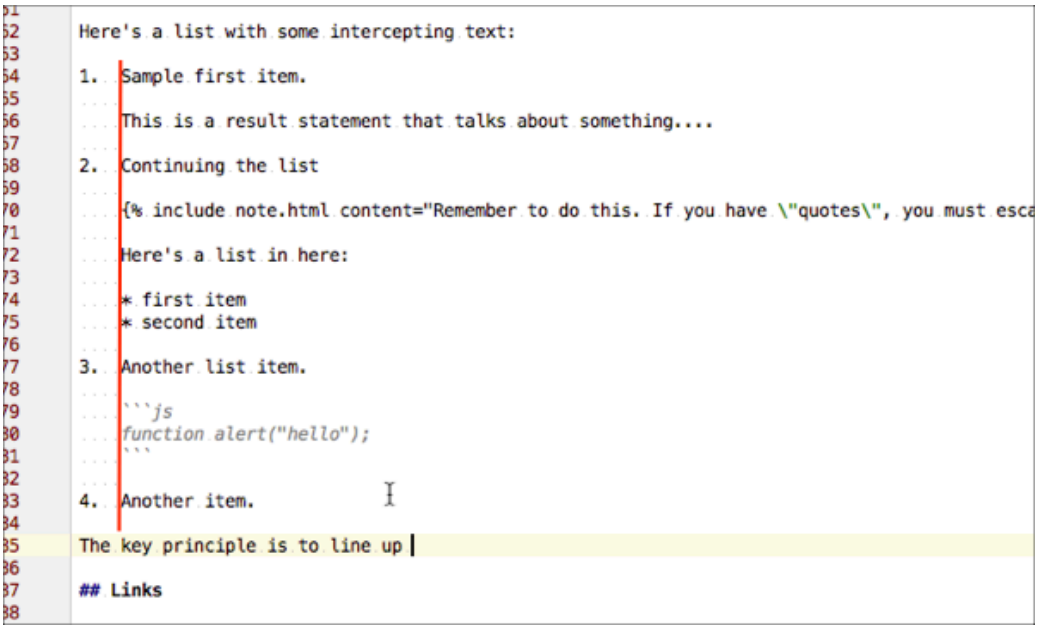
3. Another list item.

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

4. Another item.

Key Principle to Remember with Lists

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



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

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

Conditional logic

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

About Liquid and conditional statements

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

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

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

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

☑ **Tip:** Definitely check out Liquid's documentation (http://docs.shopify.com/themes/liquid-documentation/basics) for more details about how to use operators and other liquid markup. The notes here are a small, somewhat superficial sample from the site.

Where to store filtering values

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

Conditional logic based on config file value

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

```
platform: mac
```

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

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

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

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

```
To bake a casserole:

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

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

Or operator

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

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

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

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

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

This won't work.

Unless operator

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

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

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

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

Storing conditions in the _data folder

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

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

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

Specifying the location for _data

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

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

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

```
data_source: data_alpha
```

Then create a folder called _data_alpha.

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

```
data_source: data_beta
```

Then create a folder called _data_beta.

Conditions versus includes

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

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

Content reuse

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

About content reuse

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

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

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

Also, when you include a file, all of the file’s contents get included. You can’t specify that you only want a specific part of the file included. However, you can use parameters with includes. See the following Jekyll cast for more info about using parameters with includes:

Page-level variables

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

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

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

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

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

Collections

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

What are collections

Collections are custom content types different from pages and posts. You might create a collection if you want to treat a specific set of articles in a unique way, such as with a custom layout or listing. For more detail on collections, see Ben Balter's explanation of collections here (http://ben.balter.com/2016/02/20/jekyll-collections/).

Create a collection

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

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

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

Interacting with collections

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

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

How to use collections

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

Video tutorial on collections

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

WebStorm Text Editor

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

About text editors and WebStorm

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

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

Remove unnecessary plugins

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

Set default tab indent to 3 spaces instead of 4

You can set the way the tab works, and whether it uses spaces or a tab character. For details, see Code Style. JavaScript (https://www.jetbrains.com/help/webstorm/2016.1/code-style-javascript.html?origin=old_help#d658997e132) in WebStorm's help.

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

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

Add the Markdown Support plugin

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

Enable Soft Wraps (word wrapping)

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

Exclude a directory

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

Set tabs to 4 spaces

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

To set the indentation, see the "Tabs and Indents" topic in this Code Style. Javascript (https://www.jetbrains.com/help/webstorm/2016.1/code-style-javascript.html?origin=old_help#d658997e132) topic in Webstorm's help.

Shortcuts

It can help to learn a few key shortcuts:

Command	Shortcuts
Shift + Shift	Allows you to find a file by searching for its name.
Shift + Command + F	Find in whole project. (WebStorm uses the term “Find in path”.)
Shift + Command + R	Replace in whole project. (Again, WebStorm calls it “Replace in path.”)
Command + F	Find on page
Shift + R	Replace on page
Right-click > Add to Favorites	Allows you to add files to a Favorites section, which expands below the list of files in the project pane.
Shift + tab	Applies outdenting (opposite of tabbing)
Shift + Function + F6	Rename a file
Command + Delete	Delete a file
Command + 2	Show Favorites pane
Shift + Option + F	Add to Favorites

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

Finding files

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

Identifying changed files

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

Creating file templates

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

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

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

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

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

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

Disable pair quotes

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

Atom Text Editor

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

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

Customize the invisibles and tab spacing in Atom:

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

Turn off auto-complete:

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

Atom Shortcuts

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

(For Windows, replace "Cmd" with "Ctrl".)

Sidebar Navigation

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

Navgoco foundation

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

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

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

Accordion sidebar feature

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

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

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

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

Fixed position sidebar

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

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

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

Opening sidebar links into external pages

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

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

```
{% if folderitem.external_url %}
    <li><a href="{{folderitem.external_url}}" target="_blan
k">{{folderitem.title}}</a></li>```


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

## Sidebar item highlighting

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

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


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

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

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

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

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

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

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

YAML tutorial in the context of Jekyll

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

Overview

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

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

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

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

YAML overview

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

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

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

YAML basics

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

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

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

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

Example 1: Simple mapping

YAML:

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

Markdown + Liquid:

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

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

Result:

Husband's name: Tom

Wife's name: Shannon

Example 2: Line breaks

YAML:

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

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

Markdown:

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

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

Result:

Feedback

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

Block

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

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

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

Example 3: Simple list

YAML:

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

Markdown + Liquid:

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

Result:

- mountain bikes
- road bikes
- hybrid bikes

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

Example 4: List items

YAML:

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

Markdown + Liquid:

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

Result:

Regions

- US
- Spain
- France

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

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

Example 5: Table of contents

YAML:

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

Markdown + Liquid:

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

Result:

Group 1

- Thing 1
- Thing 2

- Thing 3

Group 2

- Piece 1
- Piece 2
- Piece 3

Group 3

- Widget 1
- Widget 2
- Widget 3

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

Example 6: Variables

YAML:

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

Markdown:

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

Result:

Greetings earthling!

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

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

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

Example 7: Positions in lists

YAML:

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

Markdown:

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

Result:

zero

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

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

Example 8: Properties from list items at specific positions

YAML:

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

Markdown + Liquid:

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

Result:

```
red
```

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

Example 9: Conditions

YAML:

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

Markdown + Liquid:

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

Result:

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

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

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

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

And here is the result:

- apple.com

More resources

For more examples and explanations, see this helpful post on tournemille.com: How to create data-driven navigation in Jekyll (http://www.tournemille.com/blog/How-to-create-data-driven-navigation-in-Jekyll)
.

Tags

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

Add a tag to a page

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

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

Tags overview

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

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

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

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

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

Setting up tags

Tags have a few components.

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

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

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

   Each tag archive file needs only this:

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

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

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

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

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

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

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

Retrieving pages for a specific tag

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

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

Here’s how that code renders:

Getting started pages:

- 성신여대 오픈소스 툴 실습 (page 1)
- About the theme author (page 14)
- About Ruby, Gems, Bundler, and other prerequisites (page 16)
- Install Jekyll on Mac (page 23)
- Pages (page 33)
- Posts (page 40)
- Release notes 5.0 (page 0)
- Release notes 6.0 (page 86)
- Sidebar Navigation (page 61)
- Support (page 15)
- Supported features (page 8)

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

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

Here’s how that code renders:

Getting started pages:

- About Ruby, Gems, Bundler, and other prerequisites (page 16)
- About the theme author (page 14)
- Install Jekyll on Mac (page 23)
- Pages (page 33)
- Posts (page 40)
- Release notes 5.0 (page 0)
- Release notes 6.0 (page 86)
- Sidebar Navigation (page 61)
- Support (page 15)
- Supported features (page 8)
- 성신여대 오픈소스 툴 실습 (page 1)

Efficiency

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

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

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

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

Empty tags?

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

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

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

Remembering the right tags

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

See WebStorm Text Editor (page 56) for tips on creating file templates in WebStorm.

Series

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

Using series for pages

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

1. Create the series button

First create an include that contains your series button:

```
<div class="seriesContext">
    <div class="btn-group">
        <button type="button" data-toggle="dropdown" class="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 | remove: "/"}}">{{p.weigh
t}}. {{p.title}}</a>
            </li>
            {% endif %}
            {% endif %}
            {% endfor %}
        </ol>
    </div>
</div>
```

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

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

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

2. Create the “next” include

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

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

Change “acme” to the name of your series.

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

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

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

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

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

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

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

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

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

{% include custom/series_acme.html %}

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

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

Changing the series drop-down color

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

Using a collection with your series

Instead of copying and pasting the button includes on each of your series, you could also create a collection and define a layout for the collection that has the include code. For more information on creating collections, see Collections (page 54) for more details.

Release notes 6.0

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

Relative links

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

Subfolders for pages

You can creates folders and subfolders for your pages, similar to how you can store posts in folders and subfolders. When Jekyll builds the site, all pages get pushed into the root directory as single html files (rather than being pushed inside folders, or remaining in subfolders). See Pages (page 33) for more details.

Alerts templates

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

Image templates

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

Automated links using Markdown formatting

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

Workflow maps

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

Upgrading

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