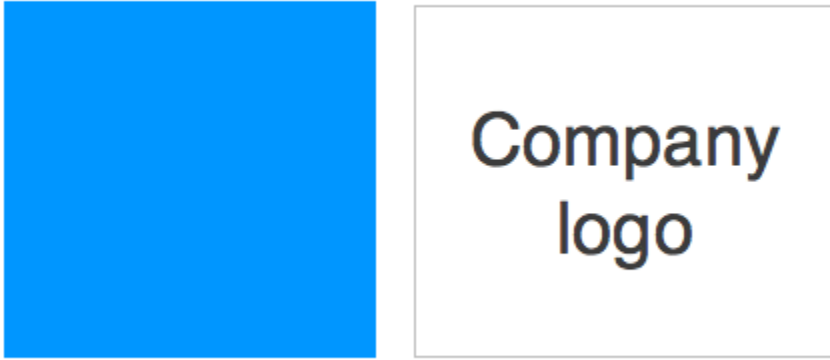


Jekyll How-to Guide

Version 1.0



Last generated: June 28, 2017

This is the copyright notice...

Table of Contents

Theme Instructions.....	5
Getting Started.....	6
Sidebar Navigation.....	10
Content and Formatting.....	15
Generate a PDF	44
Localize your Content.....	52
Troubleshooting	64
General Jekyll Topics	67
About Ruby Gems and Bundler	68
Install Jekyll on Mac.....	74
Install Jekyll on Windows.....	78
Tips for Atom Text Editor	81

Get Started

In this section:

- Getting Started 6
- Sidebar Navigation 10
- Content and Formatting 15
- Generate a PDF 44
- Localize your Content..... 52
- Troubleshooting..... 64

Theme Instructions

This theme is intended for technical documentation projects (e.g., user guides and manuals for software or hardware). This Jekyll theme uses pages exclusively and features robust multi-level navigation as well as both web and PDF output.

This version of the theme delivers the theme files through RubyGems, which means you won't see the `_includes`, `_layouts`, `_sass`, or `assets` folders in the theme's files. They're instead stored in the gem. You can pull them out of the gem if you want, but then you'll lose out on the ability to update the theme through Bundler.

- [Installation \(page 6\)](#)
- [The Location of the Theme Files \(page 7\)](#)
- [Source and gem repos \(page 7\)](#)
- [Overwriting Gem Files \(page 8\)](#)
- [Pages \(page 8\)](#)
- [Frontmatter \(page 8\)](#)

Installation

1. Install Jekyll:
 - [Install Jekyll on Mac \(page 74\)](#)
 - [Install Jekyll on Windows \(page 78\)](#)

You can check if you already have Jekyll installed by running `jekyll -v`. If you don't have the latest version, run `gem update jekyll`.

2. Install [Bundler](#) (if you don't already have it):

```
gem install bundler
```

You can check your version of bundler by running `bundler -v`. If you don't have the latest version, run `gem update bundler`.

3. Download the gem-based theme from the [documentation-theme-jekyll-multioutput-gem](#) repo.
4. Use Bundler to install or update the gem-theme's files:

```
bundle update
```

5. Configure the values in the `_config.yml` file based on the inline code comments in that file.

The Location of the Theme Files

This is a gem-based theme, which means that most of the theme's files are stored in a gem instead of inside the Jekyll project. The gem is called [documentation-theme-jekyll-multioutput](#). By storing the files in a gem, the theme files become available (and easy to update) across many different repos. Without the gem, the theme files would be hard-coded in each repo, making it difficult to apply updates when the same theme is used on many different Jekyll projects.

The theme files in the them gem contain four folders:

- `_includes`
- `_assets`
- `_layouts`
- `_sass`

To see which version of the gem you have, `cd` to your project's directory and run:

```
bundle show documentation-theme-jekyll-multioutput
```

The response will indicate the version in the path. For example, `/usr/local/lib/ruby/gems/2.3.0/gems/documentation-theme-jekyll-multioutput-0.2.0`. This is version 0.2.0 of the gem. You can compare this version against the version listed on the [gem's RubyGem page](#).

The `show` command tells you where the gem files are stored. On a Mac, you can run `open <path>`, replacing `<path>` with the `/usr/local/...` path returned above to open a Finder window showing the gem's files. On Windows, you can run `explorer <path>` to open an Explorer window.

To update the gem, run the following from your Jekyll project directory:

```
bundle update documentation-theme-jekyll-multioutput
```

Or just run `bundle update` to update all gems.

When you run this command, any updates to the theme will be pulled into your gem.

Source and gem repos

There are two repos related to this theme:

- The source repo: [documentation-theme-jekyll-multioutput](#)
- The gem-based theme repo: [documentation-theme-jekyll-multioutput-gem](#)

The source repo (documentation-theme-jekyll-multioutput) contains the theme files (including the `_layout`, `_includes`, `_sass`, and `_assets` folders) hard-coded into the Jekyll theme, without making use of the gem. This is the project source from which the gemspec

is generated (the `gemspec` is what powers the `RubyGem`). You shouldn't work with the `documentation-theme-jekyll-multioutput` repo directly unless you never want updates for your theme or don't want to use a gem-based theme.

The gem-based theme repo (`documentation-theme-jekyll-multioutput-gem`) does not include the `_layout`, `_includes`, `_sass`, and `_assets` folders because these folders are instead delivered through the gem. This repo also contains a `Gemfile` that specifies the `documentation-theme-jekyll-multioutput` gem.

Overwriting Gem Files

You can [overwrite any gem-based files](#) by adding a file by a similar name in your project. (You can also add additional files not in the gem as well.)

For example, suppose you want to overwrite the content in `_includes/logo.html`. Add a directory called `_includes` in your project, and create a file called `logo.html`. Any adjustments you make will overwrite the original `_includes/logo.html` file stored in the gem. (Most likely you would copy the content from the `_includes/logo.html` file in the gem as a basis for your modification.)

If you want to get rid of the theme entirely and make all files local, see [Converting gem-based themes to regular themes](#).

Pages

This theme uses pages exclusively. Organize your pages inside the `_docs` folder. You can store your pages in any folder structures you want, with any level of folder nesting. Assuming you have the `permalink` property correctly configured in the page's frontmatter, the site output will pull all of those pages out of their folders and put them into the root directory. This "flattening" of the page hierarchy enables relative linking in the project, which is critical when moving your content across different environments (beta, review, prod).

Having all files flattened in the root directory on site build is essential for relative linking to work and for all paths to JS and CSS files to be valid. Relative links allow you to view the built site offline or to push it from one environment or directory structure to the next without worrying about valid paths to theme assets or other links.

Frontmatter

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


```

---
title: "Sample 1: The Beginning"
permalink: sample.html
sidebar: generic
product: Generic Product
---
```

Tip: You can store the `sidebar` and `product` frontmatter as defaults in your project's `_config.yml` file.

Property	Description
title	The title for the page. If you want to use a colon in your page title, you must enclose the title's value in quotation marks. Note that titles in your pages' frontmatter are not synced with the titles in your sidebar data file. If you change it in one place, remember to change it in the other too. Use the same name as your file name, but use ".html" instead of ".md" in the permalink property. Do not put forward slashes around the permalink (this makes Jekyll put the file inside a folder in the output). When Jekyll builds the site, it will put the page into the root directory rather than leaving it in a subdirectory or putting it inside a folder and naming the file index.html.
permalink	
sidebar	The name of the sidebar that this page should use (don't include ".yml" in the name).
product	The product for this content. This appears in search and will be used as a faceted search filter at some point in the future.
toc-style: kramdown	Tells the theme you want to use the kramdown-built on-page TOC. See On-Page Table of Contents (page 17) in the Formatting topic for details how to insert the TOC using kramdown markup. This property is applied as a default in the <code>_config.yml</code> file. If you prefer the JS-generated on-page TOC instead, remove this default from your <code>_config.yml</code> file.

Sidebar

To configure the sidebar, copy the format shown in `_data/generic.yml` into a new sidebar file. Keep `generic.yml` as an example in your project, because YAML syntax can be picky and sometimes frustrating to get right. `Generic.yml` shows an example of content at every level.

Each of your pages should reference the appropriate sidebar either in the page's frontmatter or as defined in the defaults in your `_config.yml` file.

Sidebar Navigation

The output for the sidebar navigation gets generated from the sidebar yaml file in the `_data` folder. YAML files don't use markup tags but rather spacing to create syntax. Look carefully at the YAML syntax in the sample `jeekyllhowto.yml` file. The syntax for your YAML content must be correct in order for the files to be valid.

- [How the Sidebar Works \(page 10\)](#)
- [Hippo Theme \(page 10\)](#)
- [Entries in the Sidebar \(page 11\)](#)
- [Sidebar Object Hierarchy \(page 11\)](#)
- [Required properties for `folderitems` items \(page 14\)](#)
- [Adding Additional Resources \(page 14\)](#)

How the Sidebar Works

The theme contains a file called `_includes/sidebar.html` that uses “for” loops to iterate through the items in this YAML file and push the content into an HTML format. When Jekyll builds your site, the sidebar gets included into each page. This means each page has its own copy of the sidebar code when the site builds.

Hippo Theme

The theme also includes a layout for publishing to a CMS called Hippo. With this layout, the sidebar gets generated a bit differently.

For the Hippo theme, the sidebars are not pushed into each page. Instead, you manually create a separate sidebar file (Jekyll doesn't create it for you) in the `hippomenus` directory. You will upload this sidebar file into a directory on Media Central. Then each Hippo page will reference and pull in this same sidebar file.

With the Hippo layout (in `_layouts/hippolayout.html` in the gem file), code at the top of each page references a sidebar file, like this:

```
<script type="text/javascript" src="https://images-na.ssl-images-amazon.com/images/G/01/mobile-apps/devportal/includescript.min._TTH_.js"></script>
<div class="rightMenu"> <span class="inc:https://images-na.ssl-images-amazon.com/images/G/01/mobile-apps/devportal/menus/{{page.sidebar}}._TTH_.html"> </span></div>
```

The `includedescript.min.js` file is a script that pulls in an HTML file. The HTML file is dynamically specified by the `{{page.sidebar}}` variable in the file reference.

The file in the `hippo_menus` folder contains the standalone sidebar for your project on the Hippo site. When you're ready to publish your content, you will upload this sidebar into a Media Central folder.

Entries in the Sidebar

Each entry in the sidebar files includes four properties — `title`, `jurl`, `hurl`, and `ref`. These properties stand for the page title, Jekyll URL, Hippo URL, and the Markdown referent for linking. Here's an example:

```
- title: Sample1
  jurl: /sample.html
  hurl: /solutions/devices/fire-tv/docs/sample
  ref: sample1
```

Note that the `hurl` property does not use file extensions (such as `.html`), whereas the `jurl` property does. This is because Hippo is on a web server with Apache (which can handle this routing) whereas the Jekyll theme must be able to run locally without a server.

Sidebar Object Hierarchy

In addition to the properties required for each entry, sidebar entries must appear in the following hierarchy:

```

folders:
- title: Theme documentation
  folderitems:

  - title: Homepage
    jurl: /index.html
    hurl: https://developer.amazon.com/index
    ref: home

- title: Sample Folder Title
  folderitems:

  - title: Sample1 level1
    jurl: /sample.html
    hurl: https://developer.amazon.com/i/solutions/devices/fire-tv/docs/sample1-level1
    ref: sample1

    subfolders:
    - title: Another level deep
      output: web
      subfolderitems:

      - title: Some content
        jurl: /sublevel1.html
        hurl: https://developer.amazon.com/i/solutions/devices/fire-tv/docs/sample1-level2
        ref: sublevel1

      subsubfolders:
      - title: Last level deep
        output: web
        subsubfolderitems:

        - title: Some content last level
          jurl: /sublevel1.html
          hurl: https://developer.amazon.com/i/solutions/devices/fire-tv/docs/sample1-level3
          ref: sample

```

That is, there's a `folders` object contains `folderitems`. Inside `folderitems` is another level called `subfolders`, which contains `subfolderitems`. Below `subfolders` is another level called `subsubfolders`, which contains `subssubfolderitems`. Each new level begins on a new line two spaces out from the previous one. The hierarchy looks like this:

```

folders:
  folderitems:
    subfolders:
      subfolderitems:

```

Don't change any of these object names that indicate the levels. The theme's template files use a `for` loop to iterate through this structure based on these object names.

You must have items at each level. If you want to have a folder contain other folders and no individual items, you must add a `-` at that level. For example:

```

folders:

- title: My parent folder
  folderitems:

    -

      subfolders:
        - title: My child folder
          output: web
          subfolderitems:

            - title: Some content
              jurl: /sublevel1.html
              hurl: https://developer.amazon.com/i/solutions/devices/fire-t
v/docs/sample1-level2
              ref: sublevel1

            - title: My child folder
              output: web
              subfolderitems:

                - title: Some content
                  jurl: /sublevel1.html
                  hurl: https://developer.amazon.com/i/solutions/devices/fire-t
v/docs/sample1-level2
                  ref: sublevel1

```

Tip: When you're creating new levels, it's easiest to copy the correct formatting and then adjust the values. Use the sample formatting included in the generic.yml file to copy and paste new levels. If you get the spacing wrong, when Jekyll builds the project, it will usually throw an error and note a mapping problem in your YAML file.

Required properties for `folderitems` items

Each item that appears under `folderitems` must have these properties:

PropertyDescription

<code>title</code>	The page title.
<code>hurl</code>	The Hippo URL to your content. (This is only required if you're publishing to Hippo.)
<code>jur</code>	The Jekyll URL. Use a relative link that begins with a <code>/</code> and includes only page's filename, not the folders. Use the ".html" file extension (even if your file has an .md extension in the source).
<code>ref</code>	The shortname used to create the Markdown link references. This is a friendly way to refer to the page. You use this value as the Markdown referent when inserting links in your content.

Why not just have one link? Why include both `hurl` and `jur`? Here's the reason:

- Links in Hippo must be *absolute*, not *relative*.
- Links in a standalone Jekyll site that is viewed locally on your computer and on other locations must be *relative*.

The `ref` property is like a variable used to refer to either the `hurl` or `jur` property, depending on which configuration file you use when building your Jekyll project.

Adding Additional Resources

If you want to add some additional resources, such as to forums or other documentation, you can add them in a Related Resources section below the sidebar.

Here's an example:

```
#####

related_resources_title: Other Resources

related_resources_list:

- title: Forums
  hurl: https://some.developer.forum.com/

- title: More Documentation
  hurl: https://more.documentation.company.com

- title: Documentation Portal
  hurl: https://my.documentation.portal
```

Content and formatting

This page covers all the details you need to know about content and formatting, including topics such as page directories, links, alerts, images, and more.

- [Where to Store Your Pages \(page 16\)](#)
- [Pages and Front matter \(page 16\)](#)
- [Markdown Formatting \(page 17\)](#)
- [On-Page Table of Contents \(page 17\)](#)
- [Headings \(page 17\)](#)
- [Second-level heading \(page 17\)](#)
 - [Third-level heading \(page 17\)](#)
 - [Fourth-level heading \(page 18\)](#)
- [Second level header \(page 18\)](#)
- [Bulleted Lists \(page 18\)](#)
- [Numbered list \(page 18\)](#)
- [Complex Lists \(page 19\)](#)
- [Another Complex List \(page 19\)](#)
 - [Key Principle to Remember with Lists \(page 20\)](#)
- [Alerts \(page 21\)](#)
- [Callouts \(page 22\)](#)
- [Using Variables Inside Parameters with Includes \(page 23\)](#)
- [Links \(page 24\)](#)
 - [Cross-References \(page 24\)](#)
 - [Bookmark Links on the Same Page \(page 25\)](#)
 - [Links to Sections on Other Pages \(page 25\)](#)
 - [Links to External Web Resources \(page 26\)](#)
- [Detecting Broken Links \(page 27\)](#)
- [Detecting broken links across the entire site \(page 28\)](#)
- [Content re-use \(includes\) \(page 28\)](#)
- [Variables \(page 28\)](#)
- [Audio Includes \(page 29\)](#)
- [Single sourcing \(page 29\)](#)
- [Code Samples \(page 29\)](#)
- [Markdown Tables \(page 31\)](#)
- [HTML Tables \(page 32\)](#)
- [One-off Styles \(page 34\)](#)
- [Images \(page 35\)](#)
- [Excluding Images from Translated Builds \(page 36\)](#)
- [Including Inline Images \(page 36\)](#)
- [Bold, Italics \(page 37\)](#)
- [Question and Answer formatting \(page 37\)](#)
- [Glossary Pages \(page 38\)](#)

- [Tooltips \(page 39\)](#)
- [Navtabs \(page 40\)](#)

Where to Store Your Pages

Store your files the `_docs` folder, inside a project folder that reflects your product's name. Inside your project folder, you can organize your pages in any of subdirectories you want. As long as each page has a `permalink` property in the front matter, the pages will be moved into the root directory and flattened (that is, pulled out of any subdirectories) when your site builds.

Pages and Front matter

Each Jekyll page (which uses an `.md` extension) has front matter at the top set off with three hyphens at the top and bottom. The front matter for each page should look like this:

```
---
title: My File Name
permalink: myfile.html
sidebar: mysidebar
product: My Product
---
```

You can store the `sidebar` and `product` properties as defaults in your `_config.yml` file if you want. See the `defaults` property there.

If you have a colon in your title, put the title's value in parentheses, like this:

```
---
title: "ACME: A generic project"
permalink: myfile.html
sidebar: mysidebar
product: My Product
---
```

The `layout` property for the sidebar is specified in the configuration file's defaults. `_config.yml` specifies a Jekyll layout (`default.html`).

The format for any content in the front matter must be in YAML syntax. You can't use Liquid or other `{{ }}` syntax in your front matter. (In other words, no variables in YAML.)

The `permalink` should match your file name exactly, and it should include the html file extension (even if your file is markdown).

Markdown Formatting

Jekyll uses [kramdown-flavored Markdown](#). You can read up more on kramdown and implement any of the techniques available. Some templates for alerts and images are available.

On-Page Table of Contents

To add a table of contents in your topic, add this formatting where you want the table to appear:

```
* TOC
{:toc}
```

Additionally, add this into your frontmatter:

```
toc-style: kramdown
```

If you don't have `toc-style: kramdown` in your frontmatter, the TOC won't show up in the layout.

Headings

Use pound signs before the heading title to designate the level. *Note that headings must have one empty line before and after the heading.*

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

Result:

Second-level heading

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

Result:

Third-level heading

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

Result:**Fourth-level heading**

You can also use the [Setext style headers](#) if you want. If you're converting content to Markdown from Word docs using Pandoc, Pandoc will use the Setext style header markup. This means level 2 headers will be underlined rather than containing the `##` markup:

```
Second level header
-----
```

Result:**Second level header**

First level headers are underlined with an equals sign (but since h1 headings are used only for the doc title, not any subheadings within the doc, you won't see them). Levels beyond 2 use the regular pounds signs (`###`) for markup.

Bulleted Lists

This is a bulleted list:

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

Use two spaces after the asterisk.

Result:

- first item
- second item
- third item

Numbered list

This is a simple numbered list:

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

Use two spaces after each numbered item (until number 10, then use 1 space). You can use whatever numbers you want — when the Markdown filter processes the content, it will correctly sequence the list items.

Result:

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

You can control numbering with this syntax on the line preceding a list item:

```
{:start="3"}
```

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
 1. sub-list numbered one
 2. 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

```
{% include note.html content="This is a sample note. If you have  
\"quotes\", you must escape them." %}
```

Here's a list in here:

- * first item
- * second item

3. Another list item.

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

4. Another item.

Result:

1. Sample first item.

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

2. Continuing the list

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

Here's a list in here:

- first item
- second item

3. Another list item.

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

4. Another item.

Key Principle to Remember with Lists

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

```

51 Here's a list with some intercepting text:
52
53
54 1. Sample first item.
55
56 This is a result statement that talks about something....
57
58 2. Continuing the list
59
60 {% include note.html content="Remember to do this. If you have \"quotes\", you must escape
61
62 Here's a list in here:
63
64 * first item
65 * second item
66
67 3. Another list item.
68
69 ``js
70 function alert("hello");
71 ``
72
73 4. Another item.
74
75 The key principle is to line up |
76
77 ## Links
78

```

Lining up the left edge ensures the list stays intact.

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 on this left edge, the list will not render correctly.

Alerts

For alerts, use the alerts templates, like this:

```
{% include note.html content="This is a note." %}
```

Result:

Note: This is a note.

```
{% include tip.html content="This is a tip." %}
```

Result:

Tip: This is a tip.

```
{% include warning.html content="This is a warning." %}
```

Result:

Warning: This is a warning.

```
{% include important.html content="This is important." %}
```

Result:

Important: This is important.

Alerts have just one include property: `content`. If you need to use quotation inside the `content` quotation marks, escape the quotation marks by putting back slashes (`\`) before them.

```
{{{% include warning.html content="This is a \"serious\" warning." %}}
```

Result:

Warning: This is a “serious” warning.

Note that you can use Markdown syntax inside of your alerts. (You don’t need to add `markdown="span"` tags anywhere, since they’re already included in the alert templates.)

Callouts

Callouts are similar to alerts but are intended for longer text. A callout simply has a left border that is a specific color. The color uses Bootstrap’s classes.

```
{% include callout.html content="This is my callout. It tends to be a  
bit longer, and provides less visual attention than an alert. <br/><b  
r/>Here is a new paragraph." type="info" title="Sample Callout" %}
```

Sample Callout

This is my callout. It tends to be a bit longer, and provides less visual attention than an alert.

Here is a new paragraph.
Parameters are as follows:

Property	Description	Required
<code>content</code>	The content for the parameter.	Required
<code>type</code>	The color for the callout. Options are info, warning, danger, success, primary, default.	Required
<code>title</code>	A title for the callout. The color matches the type property.	Required

As with alerts, you can use Markdown inside of callouts.

Using Variables Inside Parameters with Includes

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

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

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

In my site configuration file, I have a property called `myvariable`.

```
myvariable: ACME
```

I want to use this variable in my note include.

First, before the note, capture the content for the note's include like this:

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

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

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

Result:

Note: The company is pleased to announce an upcoming release.

Note the omission of quotation marks with variable parameters.

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

Links

There are several types of links:

- [Cross-References \(page 24\)](#)
- [Bookmark Links on the Same Page \(page 25\)](#)
- [Links to External Web Resources \(page 26\)](#)
- [Links to Sections on Other Pages \(page 25\)](#)

Cross-References

To link one documentation topic to another inside the same project (internal cross references, not links to external web resources), don't use manual Markdown links. Instead, use an automated ref property that is generated from the `_include/links.html` file (which loops through your sidebar and gets all the `ref` properties).

This automated approach is more efficient and easier than manual Markdown link formatting. Additionally, it is the only way to scale link paths for translation projects.

For each item in your sidebar menu, include a `ref` property like this:

```
- title: Sample Topic
  jurl: /sample.html
  hurl: /solutions/devices/product/docs/sample
  ref: sample
```

Then open your `_config.yml` file and make sure your project's sidebar name is included in the `sidebars` property.

The file that generates the links (`_includes/links.html`) iterates through the sidebar data files (all the ones listed in your configuration file, that is) and constructs a list of Markdown reference-style links.

(The forward slash (`/`), which is listed in the sidebar data file's `jurl` property, gets removed from the Jekyll links, which allows links to be relative. It's included in the sidebar data file to facilitate menu highlighting.)

On each of your pages, you must include the `links.html` file at the bottom of the content:

```
{% include links.html %}
```


Note: If your links don't work, check to see whether you remembered to include the `links.html` file at the bottom of each topic.

When you add the `{% include links.html %}` reference at the end of the topic, it's the equivalent of adding a [Markdown reference-style links](#) like this:

```
[sample]: somelink.html
```

You won't actually see this referent on your page because it all happens in the build process. (The `links.html` file dynamically builds all the `ref` instances and then inserts this content at the bottom of the page, and then the Markdown filter process the content, converting it to HTML and inserting links where the references appear.)

Tip: When you choose the `ref` values in your sidebar file, use the same names as your files. Otherwise it gets confusing to try to match up ref values with the right files.

You can see the output of `links.html` by looking at the `linkstest.html` file in the `_site` directory after your site builds. It should include your pages in Markdown reference style formatting. This is what gets inserted at the bottom of every page when you build your Jekyll site.

Bookmark Links on the Same Page

If you want to link to a heading on the same page, first add an ID tag to the header like this:

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

Then reference it with a normal Markdown link:

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

Result:

[Some link \(page 0\)](#)

Links to Sections on Other Pages

Suppose you want to link to a specific section heading on another page.

First create a heading ID for the section you want to link to. On the [Getting started page \(page 6\)](#), there are some headings like this:

```
## My updates {#updates}  
  
## Text editors {#editors}
```

Now add a `bookmarks` property to the entry in the sidebar data file and include all the heading ID tags on that page that you want to link to inside brackets.

```
- title: My Page Name  
  jurl: /acme-mypage-name.html  
  hurl: /some/long/path/acme-mypage-name  
  ref: acme-mypage-name  
  bookmarks: [updates, editors]
```

The `links.html` file will automatically create Markdown link references for any strings in the `bookmarks` array.

Use the following syntax for your link Markdown referent:

```
Here's a list of [editors you can use][acme-mypage-name#editors].
```

(The syntax actually resembles the same syntax for bookmark links, though the link is actually just a string.)

The `links.html` file will create references that look like this in the build:

```
[getting-started#editors]: getting-started.html#editors
```

Result:

Here's a list of [editors you can use][getting-started#editors].

Links to External Web Resources

For links to external web resources, just use regular Markdown style links using an absolute URL:

```
See the [Android documentation](https://developer.android.com/index.html).
```

Result:

See the [Android documentation](https://developer.android.com/index.html).

If links to external resources clutter the text, you can use Markdown reference style links instead of putting the URLs inline.

Example:

See the `[ColorFilterDimmer][cfdim]` class and the `[codim][ColorOverlayDimmer]` class in the Android documentation.

...

`[cfdim]: https://developer.android.com/reference/android/service/vr/VrListenerService.html`

`[codim]: https://developer.android.com/reference/android/support/v17/leanback/graphics/ColorOverlayDimmer.html`

Result:

See the [Android documentation \(page 0\)](#).

Tip: If the link formatting doesn't render correctly in your output, something is wrong with the link. Check to make sure you included the `links.html` file at the bottom of the file, and that your referent is correct.

Detecting Broken Links

If you have an error with your Markdown link reference, kramdown won't process the link. For example, suppose you referred to the link like this:

See the `[instruction for image borders][image-borders]`.

In this example, let's say the real `ref` value is `imageborders` (without the hyphen). As a result, putting `image-borders` will result in no link.

Result:

See the `[instruction for image borders][image-borders]`.

To check for broken links in your output, do a search for `[]` in your `_site` directory (restricting the search to `*.html` files only in the `_site` directory). The `[]` is a unique syntax that is unlikely to be used in many other places, and can indicate a broken link.

If you find a broken link, here are main causes:

- You forgot to add the `{% include links.html %}` at the bottom of the file.
- The Markdown referent you're using doesn't match the `ref` name in your sidebar data file.

Detecting broken links across the entire site

To check for broken links across the entire site, use the [Broken Link Checker tool](#). For URLs that are listed as containing broken links, go to the page. Then use the [Check My Links](#) Chrome extension to identify the broken link on the page.

Content re-use (includes)

To re-use content, store the content in your `_includes` folder inside the `content` subfolder:

Then use the `include` tag to reference the file. Here's an example:

```
{% include content/myfile.md %}
```

Content stored in `_includes` will be available to include in any page. To avoid naming conflicts with any includes in the gem-theme, include your product's name in the file name of your include.

Note that if you have an include that is only included in the same folder, you can use the `include_relative` tag and then put the included file in the same folder (rather than storing it in `_includes`). However, the `include_relative` tag can't reference a file that is stored outside of the folder (with `../` syntax). (You can reference subfolder locations, though.)

Variables

To use a variable, add the variable and its value to your config files, like this:

```
myvariable: ACME
```

Then reference the property through the `site` namespace:

```
{{site.myvariable}}
```

Result:

All properties in your configuration files are available through the `site` namespace. (Note that if you add values to your configuration files, you must restart Jekyll for the changes to take effect.)

All properties in the page's front matter are available through the `page` namespace.

If you have a lot of variables, you could also store them in the `_data` folder (for example, `_data/myvars.yml`). Then you would reference them through the `site.data` namespace (for example, `site.data.myvars.myvariable`).

Audio Includes

If you have an audio file that you want to include, you can use the audio include. Audio includes work similar to images. Here's an example:

```
{% include audio.html title="Example: Basic Punctuation" file="jekyllhowto/audio/great" type="mp3" %}
```

The parameters of the include are as follows:

Parameter	Description
title	A title tag for the element. Optional. This tag might be useful for SEO, but the title does not appear anywhere in the audio player's display.
file	The name of the audio file, without the file extension.
type	The extension for the file.

Single sourcing

Suppose you have content that you want to push out to multiple files. But there are some differences that each destination page should have.

Create the include as usual. Where you want to differentiate the content, you could add this:

```
{% if include.device == "product_a" %}  
Say this for product A only...  
{% endif %}
```

When you call the include, pass this parameter into the include:

```
{% include content/{{site.language}}/myfile.md device="product_a" %}
```

Code Samples

For code samples, use fenced code blocks with the language specified, like this:

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

Result:

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

For the list of supported languages you can use and the official abbreviations, see [Supported languages](#).

Jekyll applies syntax highlighting using a stylesheet that color codes the text based on the language.

If you want to make specific text red inside a code sample (leaving all other text black), use `pre` tags instead of backticks, and then use `` tags inside the code.

For example, suppose you want to call attention to a particular line in a code example, in this case, `console.log`. You can apply a `red` class to that content to make it more apparent:

```
<pre>
if (chocolate == "healthy") {
  chocolate = chocolate + 10000;
  <span class="red">console.log("chocolate healthy: " + chocolat
e);</span>
}
else (chocolate == "unhealthy") {
  chocolate = chocolate + 50000;
  <span class="red">console.log("chocolate unhealthy: " + chocolat
e);</span>
}
</pre>
```

Result:

```
if (chocolate == "healthy") {
  chocolate = chocolate + 10000;
  console.log("chocolate healthy: " + chocolate);
}
else (chocolate == "unhealthy") {
  chocolate = chocolate + 50000;
  console.log("chocolate unhealthy: " + chocolate);
}
```

Note that double curly braces `{{ }}` are reserved characters, so you cannot actually use them in code samples. If you have double curly braces, surround them with `raw` tags like this:

```
{% raw %}{{ }}{% endraw %}
```

Jekyll will not process any logic inside of `raw` tags.

If your code sample is XML, this approach using `<pre>` tags won't be enough. You'll need to [escape all the HTML](#) and leave the `` tags unescaped.

Note that currently there's a bug with Liquid when using the `highlight` tag for code samples within lists. If you run into this issue, use the fenced code block (with backticks) instead of using the `highlight` tag.

Markdown Tables

You can use standard Markdown syntax for tables:

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

Result:

Priority applesSecond priorityThird priority

| | | |
|------------|--------------|---------------|
| ambrosia | gala | red delicious |
| pink lady | jazz | macintosh |
| honeycrisp | granny smith | fuji |

However, Markdown tables don't give you control over the column widths. Additionally, you can't use block level tags (paragraphs or lists) inside Markdown tables, so if you need separate paragraphs inside a cell, you must use `

`.

If you want to add a class to the table in Markdown, add a tag like this:

```
{: .mystyle}
Priority apples	Second priority	Third priority
ambrosia	gala	red delicious
pink lady	jazz	macintosh
honeycrisp	granny smith	fuji
```

This will create a table with a class of `mystyle`.

You could then add an embedded style for `mystyle`:

```
<style>

.mystyle th {
  font-weight: bold;
}
</style>
```

HTML Tables

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


```

<table>
<colgroup>
<col width="60%" />
<col width="40%" />
</colgroup>
<thead>
<tr>
<th>To create...</th>
<th>Use this skill type</th>
</tr>
</thead>
<tbody>
<tr>
<td markdown="span">
A skill that can handle just about any type of request.

```

For example:

- Look up information from a web service
- Integrate with a web service to order something (order a car from Uber, order a pizza from Domino's Pizza)
- Interactive games
- Just about anything else you can think of

```

</td>
<td markdown="span">
Custom skill (*custom interaction model*)

```

See [Sample 3][sample3]

```

...(more content...)
</td>
</tr>
<tr>
<td markdown="span">
...(content in second row, first column)
</td>
<td markdown="span">
...(content in second row, second column)
</td>
</tr>
</tbody>
</table>

```

Result:

To create...

A skill that can handle just about any type of request.

For example: - Look up information from a web service

- Integrate with a web service to order something

(order a car from Uber, order a pizza from Domino's

Pizza) - Interactive games - Just about anything else

you can think of

...(content in second row, first column)

Use this skill type

Custom skill (*custom interaction model*) See [Sample 3 \(page 0\)](#)

...(more content...)

...(content in second row, second column)

To make life easier, add the following into a template that you can easily trigger through Atom's snippet feature:

```
<table>
  <colgroup>
    <col width="40%" />
    <col width="60%" />
  </colgroup>
  <thead>
    <tr>
      <th></th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td></td>
      <td></td>
    </tr>
    <tr>
      <td></td>
      <td></td>
    </tr>
  </tbody>
</table>
```

One-off Styles

If you have a need to implement a custom style within your Markdown content, first define the style through some style tags:

```
<style>
.special {
  font-family: Gothic;
  font-size: 40px;
  font-color: red;
}
</style>
```

Then apply the class with this syntax:

```
{: .special}
My special class.
```

Result:

My special class.

You can also use ID tags instead of classes:

```
{: #special}
My special class.
```

You can have an empty line between the class or ID tag (`{: #special}`) and the next element. This technique will apply the class or ID attribute on whatever element comes next in your document.

Images

To insert an image into your content, use the image.html include template that is set up:

```
{% include image.html file="jekyllhowto/images/company_logo" type="png" url="http://developer.company.com" alt="My alternative image text" caption="This is my caption" border="true" max-width="90%" %}
```

Result:



This is my caption

The image include's properties are as follows:

Property	Description	Required?
----------	-------------	-----------

<code>file</code>	The name of the file (without the file extension)	Required
-------------------	---	----------

Property	Description	Required?
<code>type</code>	The type of file (png, svg, and so on)	Required
<code>url</code>	Whether to link the image to a URL	Optional
<code>alt</code>	Alternative image text for accessibility and SEO	Optional
<code>caption</code>	A caption for the image	Optional
<code>border</code>	A border around the image. If you want the border, set this equal to <code>true</code> . Otherwise omit the parameter.	Optional
<code>max-width</code>	You can use px or a percentage, such as <code>70px</code> .	Optional

The image template will use the `image_path` property when referencing the path to the image.

Store images in the **images** folder in your Jekyll project — these images will be used for your Jekyll output.

Media Central will cache images you upload and expire the cache on an *hourly* basis. The first time you upload an image to Media Central, you may need to wait a few minutes before it becomes available.

Excluding Images from Translated Builds

If you want to have some images appear only in certain languages, use conditional logic:

```
{% if site.language == "english" %}

{% include image.html file="jekyllhowto/images/company_logo" type="png" url="http://dev.company.com" alt="My alternative image text" border="true" caption="This is my caption" %}

{% endif %}
```

Including Inline Images

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

```
Click the Android SDK Manager button {% include inline_image.html file="jekyllhowto/images/androidsdkmanagericon" type="png" alt="SDK button" border="true" max-width="90%" %}
```

Result:

Click the **Android SDK Manager** button 

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

Property	Description	Required
<code>file</code>	The name of the file	Required
<code>type</code>	The type of file (png, svg, and so on)	Required
<code>alt</code>	Alternative image text for accessibility and SEO	Optional
<code>border</code>	A border around the image. If you want the border, set this equal to <code>true</code> . Otherwise omit the parameter.	Optional
<code>max-width</code>	A maximum width for the image. You can use px or a percentage, such as <code>70px</code> .	Optional

Bold, Italics

You can make content **bold** with two asterisks (`**bold**`), or *italics* with one asterisk (`*italics*`).

Question and Answer formatting

You can use the following formatting for Q&A pages:

```
Why is the sky blue?
: It's not actually blue. This is an illusion based on the way molecules in our atmosphere reflect light waves.

Why is the ocean blue?
: It reflects the color of the atmosphere.
```

Result:

Why is the sky blue?

It's not actually blue. This is an illusion based on the way molecules in our atmosphere reflect light waves.

Why is the ocean blue?

It reflects the color of the atmosphere.

If you want to emphasize the question aspect, put a “Q:” before the questions:

```
Q: Why is the sky blue?
: It's not actually blue. This is an illusion based on the way molecules in our atmosphere reflect light waves.

Q: Why is the ocean blue?
: It reflects the color of the atmosphere.
```

Result:

Q: Why is the sky blue?

It's not actually blue. This is an illusion based on the way molecules in our atmosphere reflect light waves.

Q: Why is the ocean blue?

It reflects the color of the atmosphere.

kramdown outputs this Markdown syntax as a definition list in HTML.

Glossary Pages

To list the terms from a glossary, first list the terms in a YAML data file inside the `_data` folder, like this:

```
-
  term: macabre
  def: ghastly, horrifying, resembling death

-
  term: riparian
  def: on the bank by a river
```

Supposing the glossary file were named **glossary.yml**, you could list out the terms like this:

```
{% assign glossaryTerms = site.data.glossary %}

<dl>
  {% for entry in glossaryTerms %}
    <dt id="{{entry.term}}">{{entry.term}}</dt>
    <dd>{{entry.def}}</dd>
  {% endfor %}
</dl>
```

Result:

The terms will be sorted according to their order in the glossary.yml file, so you have to manually alphabetize the terms. (Liquid's `sort` filter doesn't mix capitalized and lowercased terms when alphabetizing items, so you can't use it here unless you only have lowercase or only have uppercase terms.)

Each definition term will have an ID tag (based on `id="{{entry.term}}"` in the previous code). If you want to link to this term, add each glossary term to the `bookmarks` property in your glossary entry in your sidebar navigation. For example:

```
- title: Glossary
  jurl: /glossary.html
  hurl: /solutions/devices/glossary
  ref: glossary
  bookmarks: [mcabre, riparian]
```

Then make links in your content like this:

Result:

For more information, see [Riparian][glossary#riperian].

Tooltips

You can leverage your glossary for tooltips by using the tooltips.html include. Here's an example:

to say the least.

I cannot believe she described this canal trail as {% include tooltip
s.html term="riperian" %} in the book.

Result:

The setting was [Riparian](#) to say the least.

I cannot believe she described this canal trail as [riparian](#) in the book.

By default, the glossary term is lower-cased. If you want it capitalized, add a `capitalize="true"` parameter in the include syntax.

Note the following about links and formatting in the tooltips.yml file:

- You can't add hyperlinks in YAML content using the approach for automated links (such as `[jekyllhowto-publishing][jekyllhowto-publishing]`), but you can directly code HTML links here.
- In your link formatting, use single quotes instead of double quotes. For example, `River in Wikipedia`.
- For multiple paragraphs, use `<p>` tags. Other HTML formatting is also allowed.
- You can't use includes or variables in your glossary.yml file.
- Enclose the `def` values in quotation marks to avoid conflicts with colons, which are illegal characters in YAML syntax. For quotation marks inside quotations, escape them `/"like this/"`.

To implement the tooltip on a page, reference it through the tooltips.html include:

The parameters of the tooltips include are as follows:

ParametersDescription

term	The glossary term in the glossary.yml file
capitalize	Include only if you want the term capitalized. If so, set it equal to <code>true</code> and treat as a string. (A capitalization filter gets placed on the glossary term.) If you omit the term, no capitalization filter gets applied to the term. If the term is capitalized, you don't need to also apply this filter.

Navtabs

You can implement nav tabs when you have different code samples or instructions based on programming languages or platforms, and you want to put the information in a more compressed space. Here's the HTML code:


```

<ul id="profileTabs" class="nav nav-tabs">
  <li class="active"><a class="noCrossRef" href="#firsttab" data-toggl
e="tab">First Tab</a></li>
  <li><a class="noCrossRef" href="#secondtab" data-toggle="tab">Secon
d Tab</a></li>
  <li><a class="noCrossRef" href="#thirdtab" data-toggle="tab">Third T
ab</a></li>
  <li><a class="noCrossRef" href="#fourthtab" data-toggle="tab">Fourt
h Tab</a></li>
</ul>
<div class="tab-content">
  <div role="tabpanel" class="tab-pane active" id="firsttab">
    <div class="subheading">First Tab
    </div>
    <p>Lorem Ipsum is simply dummy text of the printing and typesettin
g industry. Lorem Ipsum has been the industry's standard dummy text ev
er since the 1500s, when an unknown printer took a galley of type and
scrambled it to make a type specimen book. It has survived not only fi
ve centuries, but also the leap into electronic typesetting, remainin
g essentially unchanged. It was popularised in the 1960s with the rele
ase of Letraset sheets containing Lorem Ipsum passages, and more recen
tly with desktop publishing software like Aldus PageMaker including ve
rsions of Lorem Ipsum.</p>

  </div>
  <div role="tabpanel" class="tab-pane" id="secondtab">
    <div class="subheading">Second tab
    </div>
    <p>Lorem Ipsum is simply dummy text of the printing and typese
tting industry. Lorem Ipsum has been the industry's standard dummy tex
t ever since the 1500s, when an unknown printer took a galley of type
and scrambled it to make a type specimen book. It has survived not onl
y five centuries, but also the leap into electronic typesetting, remai
ning essentially unchanged. It was popularised in the 1960s with the r
elease of Letraset sheets containing Lorem Ipsum passages, and more re
cently with desktop publishing software like Aldus PageMaker includin
g versions of Lorem Ipsum.</p>
  </div>
  <div role="tabpanel" class="tab-pane" id="thirdtab">
    <div class="subheading">Third tab
    </div>
    <p>Lorem Ipsum is simply dummy text of the printing and typese
tting industry. Lorem Ipsum has been the industry's standard dummy tex
t ever since the 1500s, when an unknown printer took a galley of type
and scrambled it to make a type specimen book. It has survived not onl
y five centuries, but also the leap into electronic typesetting, remai
ning essentially unchanged. It was popularised in the 1960s with the r

```

```

    release of Letraset sheets containing Lorem Ipsum passages, and more re
    cently with desktop publishing software like Aldus PageMaker includin
    g versions of Lorem Ipsum.</p>

```

```

</div>

```

```

<div role="tabpanel" class="tab-pane" id="fourthtab">

```

```

  <div class="subheading">Fourth Tab

```

```

  </div>

```

```

    <p>Lorem Ipsum is simply dummy text of the printing and typese
    tting industry. Lorem Ipsum has been the industry's standard dummy tex
    t ever since the 1500s, when an unknown printer took a galley of type
    and scrambled it to make a type specimen book. It has survived not onl
    y five centuries, but also the leap into electronic typesetting, remai
    ning essentially unchanged. It was popularised in the 1960s with the r
    elease of Letraset sheets containing Lorem Ipsum passages, and more re
    cently with desktop publishing software like Aldus PageMaker includin
    g versions of Lorem Ipsum.</p>

```

```

  </div>

```

```

</div>

```

Result:

First Tab

Second Tab

Third Tab

Fourth Tab

First Tab

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

Note the following:

- If you're adding more tabs, be sure to customize the `id` value for each `tabpanel` div to match up with the `href` value for the list item (`li`) classes. In this code, you can see that `firsttab` in the href value matches up with the `firsttab` value for the div with the `tab-pane` class.
- You can use Markdown instead of HTML inside the `<div role="tabpanel" class="tab-pane active" id="profile" markdown="block">` by adding `markdown="block"` as an attribute. This tells kramdown to process the content as block level element with Markdown. If you just want a span element, use `markdown="span"`.
- Don't use heading levels (such as `h2`) within the tabs. If you do, the heading levels will appear in the mini-TOC and the links won't jump to anywhere. Instead, use a `subheading` class on a `div` tag as shown in the example.

- You can store the tab content in a separate file and pull it in. For example, you might store the tab content for the first tab in a file called `firsttab.yml`. Store this tab in the same directory as your file with the `navtab`. Now reference the content with this: `{% include_relative firsttab.yml %}`.

If you have a code sample inside a `navtab`, put the code flush against the left edge and inside a surrounding element that specifies `markdown="block"`. Here's an example:

```
<p markdown="block">

```json
"notificationInfo": {
 "notificationType": "OrderPlacedNotification",
 "lwaClientId": "amzn1.application-oa2-client.6b68xxxxxxxxxx9",
 "notificationTime": "2016-12-02T21:09:58.689Z",
 "notificationId": "amzn1.dash.notification.v1.xxxxxxxxxxxxxx13",
 "version": "2015-06-05"
}
```

</p>
```

Generating a PDF

You can generate a PDF of your Jekyll project. The PDF uses Prince XML to generate the PDF and allows you to configure which pages you want printed. The PDF output includes a table of contents for the entire guide, a mini-TOC on each section page, page numbers in cross references, and running headers and footers. The styling uses Bootstrap's CSS for print styles. You can see a sample here: [Jekyll How-to Guide \(PDF\) \(page 0\)](#).

- [Generate a PDF of All Docs in a Sidebar \(page 44\)](#)
- [Generate a PDF of a Single Page \(page 48\)](#)
- [Modify the Print Styles \(page 48\)](#)
- [Change the PDF Styles \(page 49\)](#)
- [Change the PDF's Headers or Footers \(page 49\)](#)
- [Troubleshooting \(page 50\)](#)
- [Analyzing the Output \(page 50\)](#)

Generate a PDF of All Docs in a Sidebar

1. Install Prince XML:
 - [MacOS instructions](#)
 - [Windows instructions](#)
2. Create a configuration file for the PDF output. Copy the `_config_pdf_jekyllhowto.yml` file and rename it to match your project's name (e.g., `_config_pdf_<myproject>.yml`).
3. Open the configuration file and customize the following values:
 - `print_title` : Appears on the PDF's title page and running header.
 - `print_subtitle` : Appears on the PDF's title page.
 - `sidebar` : Used to generate the table of contents and mini-TOC.
 - `folderPath` : The folder path to your site output (for example, `/Users/tomjoht/projects/devcomm-appstore-v2/_site`). Prince needs this absolute path to access the files.
 - `copyright_notice` : The copyright statement that you want to appear in the manual.
4. Open your sidebar data file and add the following section after `folders` :

```
folders:

- title: Frontmatter
  jurl: /frontmatter.html
  type: frontmatter
  pdf: true
  folderitems:

- title: Title page
  jurl: /pdf_title_page.html
  pdf: true

- title: Copyright page
  jurl: /pdf_copyright_page.html
  pdf: true

- title: TOC page
  jurl: /pdf_toc_page.html
  pdf: true
```

5. Add a new property for each section title and page called `pdf: true` for all the pages you want included in the PDF. Follow the example in the previous code sample. Alternatively, look at `_data/jekyllhowto.yml` file.
6. Organize your pages into subfolders by section. (Sections are the `folders` title that contains `folderitems`, or the `subfolders` title that contains `subfolderitems`, etc.)

For example, if your product nav has the folders “Getting Started” and “Configuration”, create folders inside the `_docs` folder named “Getting Started” and “Configuration”. Group the pages within those sections into those folders.

7. For each folder, add a new page that will serve as the mini-TOC for that section. Call it something that like `**minitoc_.md**`.

Tip: It might be helpful to organize your pages into subfolders by section. All files get flattened into the root directory when your site builds, so it doesn’t matter how many subfolders you use to organize your content.

8. Open up each mini-TOC file and add the following, customizing the frontmatter values for your own project:

```

---
title: Get Started
permalink: /iap-minitoc-get-started.html
sidebar: in_app_purchasing
---

{% include pdfminitoc.html %}

```

The `title` should be the same as the section title in your sidebar table of contents. The `permalink` should match your file name. The `sidebar` corresponds to your product sidebar.

Note: Be careful with your file names here. If you mistype a file name, the PDF won't build and you'll have to sort out the cause of the error later. In fact, Prince is much more exacting about only generating a PDF if all listed assets are available. You'll likely encounter some errors in the PDF generation process that stem from unavailable files or mistyped names in your sidebar data file. Think of these errors as helpful validation.

- Open up your sidebar data file (in the `_data` folder) and add a `jurl` property for each **section** entry, like this:

```

- title: Jekyll Project Setup
  jurl: /jekyllhowto-project-setup.html
  pdf: true
  folderitems:

```

Point the `jurl` value to your mini-TOC file. See the `_data/jekyllhowto.yml` file for an example.

- Open up the first file in your table of contents (after the Frontmatter section). Add `class: first` in the page's frontmatter.

```

---
title: Jekyll Project Setup
permalink: /jekyllhowto-minitoc-project-setup.html
sidebar: jekyllhowto
class: first
---

```

This property will reset the numbering so that the first page begins on "1" (after the lower-roman number numbering for the TOC and frontmatter section.)

Before running Prince, we need to build an HTML-friendly output for Prince to consume. This HTML-friendly output will apply the layout that we want reflected in the print material.

11. Create a Jekyll server shortcut file to build the PDF-friendly output that the Prince script will consume. Copy **serve_pdf_jekyllhowto.sh** and customize the file name with your own project. Open the file and customize the PDF file name to match your PDF configuration file:

```
bundle exec jekyll serve --config _config_pdf_jekyllhowto.yml
```

12. Create a build shortcut file to make it easy to run Prince to generate the PDF. Copy **build_pdf_jekyllhowto.sh** and customize the file name with your own project. Open the file and customize the PDF file name (change **jekyllhowto.pdf**):

```
echo "Building the PDF ...";  
prince --javascript --input-list=_site/assets/prince-list.txt  
-o pdf/jekyllhowto.pdf;  
echo "Done. Look in the /pdf folder in your project directory.";
```

The prince-list.txt file contains scripts that iterate through your sidebar data file and gather links to all the pages to consolidate in the PDF. The **-o** parameter specifies the file name and location Prince should write the PDF file to. If you're having trouble with pages appearing, you can check prince-list.txt in your **_site/assets** output and make sure a valid link to the file exists on the page. The **--input-list** parameter in the above command is the input source for Prince.

Building the PDF

1. From the command line, build the HTML output that uses your PDF configuration file:

```
. serve_pdf_jekyllhowto.sh
```

(Use the custom Jekyll server shortcut you created earlier.)

2. When the server preview finishes, open another tab and build the PDF:

```
. build_pdf_jekyllhowto.sh
```

You don't need to have Jekyll server running to build the PDF. But it's helpful in case you build the PDF and notice some issues you want to fix.

If successful, you'll see a message like this:

```
Building the PDF ...  
Done. Look in the /pdf folder in your project directory.
```

(Actually, the message will appear even if the build is unsuccessful — it just lets you know the process finished.)

Look in the root directory of your project (not the `_site` folder), look for your PDF.

Generate a PDF of a Single Page

If you just want to create a simple PDF of one page in your docs, you can skip most of the steps in the previous section. To generate a PDF of a single doc:

1. Complete steps 1 through 3 in the previous section, [Generate a PDF of All Docs in a Sidebar \(page 44\)](#).
2. Build the HTML-friendly version of the site:

```
bundle exec jekyll serve --config _config_pdf_jekyllhowto.yml
```

3. From the command line, navigate to the `_site` folder. Then run this:

```
prince --javascript jekyllhowto-content-and-formatting.html -o pdf/jekyllhowto-content-and-formatting.pdf;
```

In this case, the page being converted to PDF is `jekyllhowto-content-and-formatting.html`. Replace this with the page you want.

Modify the Print Styles

The print styles are defined in **`assets/css/pdf/printstyles.css`**. To overwrite the styles:

1. Copy the contents of **`printstyles.css`**, which is packaged in the gem.

To get the contents of the file, run `bundle show documentation-theme-jekyll-multioutput` from your Jekyll project directory. Go to the path shown and open the **`assets/css/pdf/printstyles.css`** file.

2. Copy and paste the contents of **`printstyles.css`** into a new file called **`user_defined_pdf_styles.css`**. Put **`user_defined_pdf_styles.css`** file into a folder called **`assets/css/pdf`** in your Jekyll project.

This will overwrite the **`user_defined_pdf_styles.css`** file (which is blank) in the gem's files. This gem file is a placeholder intended to accommodate custom styles. It is referenced in the PDF layout files.

3. Change the styles as desired. See [CSS Properties](#) for a list of styles supported by Prince XML.

Change the PDF Styles

Does a style not look right? Do you want to customize the headers or footers for a specific page? You can do so by modifying the print stylesheet: **assets/css/pdf/printstyles.css**.

You can simply add regular CSS here as you want. See [CSS Properties](#) on the Prince XML site for supported properties.

Change the PDF's Headers or Footers

1. Follow the instructions in the previous section, [Change the PDF Styles \(page 49\)](#).
2. In your PDF configuration file (e.g., `_config_pdf_jekyllhowto.yml`), in the **defaults** section, add a **class** as a default to your **/_docs** and **pages**. For example:

```
defaults:
-
  scope:
    path: ""
    type: pages
  values:
    layout: printpdf
    class: myclass
-
  scope:
    path: ""
    type: docs
  values:
    layout: printpdf
    class: myclass
```

This will add a default **class** property and value to your page's frontmatter, like this:

```
---
class: myclass
---
```

3. Open the **user_defined_pdf_styles.css** file and define the style like this:

```
body.myclass { page: myclass }
@page myclass {
  @top-left {
    content: " ";
  }
  @top-right {
    content: prince-script(datestamp);
  }
  @bottom-right {
    content: counter(page, lower-roman);
  }
  @bottom-left {
    content: prince-script(guideName);
    font-size: 11px;
  }
}
```

See the [Page Selectors](#) topic in the Prince XML documentation for more details.

The `datestamp` and `guideName` functions are special Prince functions defined in the `printpdf.html` layout (packaged in the `assets/pdf` folder of the gem). Other JavaScript generated content is also possible. See the [Prince XML site](#) for details.

Troubleshooting

If you see an error that Prince can't load an input file, it means one of the files in the list is incorrectly named. For example, you might see this error after running your `build_pdf_jekyllhowto.sh` command:

```
prince: /Users/tomjoht/projects/myapp/_site/jekyllhowto-minitoc-gettin
g-started.html: error: can't open input file: No such file or director
y
```

If you see this error, look to make sure the file appears in the `_site` directory and is properly named. Check the `permalink` name of the file as well as its name in the sidebar menu file (e.g., `_data/jekyllhowto.yml`).

You might also get errors if you have JavaScript or non-allowed CSS syntax in your content.

Analyzing the Output

Look at the PDF. Check the display of your tables, images, code samples, and other formatting. Look at the running header and footer, as well as the title page, table of contents, and mini-TOC pages. Does it all look good? If so, great.

If you need to conditionalize some content so that it doesn't appear in the guide, you can use an if condition like this:

```
{% unless site.format == "pdf" %}  
This won't appear in the guide....  
{% endunless %}
```

`unless` acts like a negative. You would read the above like this: Run this code *unless* site.format equals pdf.

You can also conditionalize your content using this syntax:

```
{% if site.format == "web" %}  
This won't appear in the guide....  
{% endif %}
```

The configuration file for the Jekyll and Hippo outputs contains `format: web`. The configuration file for PDFs contains `format: pdf`.

Localization

The Jekyll theme supports translation to Japanese and German. Content for each language is stored inside the `_docs_ja` or `_docs_de` folders, which are collections that have their own default attributes.

Note: Translation is applied to HTML output, not to the Markdown source. This is because WorldServer cannot separate out HTML code from regular words in Markdown files, and consequently provides incorrect word counts and cost estimates. Additionally, localization managers do not need to rebuild the Jekyll site.

- [Prepare the Files \(page 52\)](#)
 - [Where to Store the Translated Files \(page 52\)](#)
 - [Image Paths \(page 53\)](#)
 - [Content Includes \(page 54\)](#)
 - [Conditional Text \(page 54\)](#)
 - [Sidebar \(page 55\)](#)
 - [Documentation homepage \(page 55\)](#)
 - [Links \(page 56\)](#)
 - [Top Navigation Bar \(page 56\)](#)
 - [Search \(page 57\)](#)
 - [UI Strings \(page 58\)](#)
 - [Provide a Glossary \(page 58\)](#)
 - [Follow Localization Best Practices \(page 58\)](#)
- [Push the Files Through the WorldServer Workflow \(page 59\)](#)
 - [Onboard with WorldServer through TSP \(page 59\)](#)
 - [Generate the HTML Output and Zip the Files \(page 60\)](#)
 - [Submit Your Files in WorldServer \(page 60\)](#)
 - [Downloading the Translated Files \(page 61\)](#)
- [Getting the Translated Files Back into Your Project \(page 62\)](#)
 - [Publishing on Hippo \(page 62\)](#)
 - [Publishing Updates \(page 63\)](#)

Prepare the Files

This first section explains how to properly prepare your files for translation. The next section explains how to [push your files through WorldServer's workflow \(page 59\)](#).

Where to Store the Translated Files

In Jekyll, a “collection” is a content type (similar to pages) with custom attributes and values that you can define.

In the Jekyll theme, each language has its own collection for storing Markdown files, images, and other assets. The three language collections are as follows:

- `_docs`
- `_docs_ja`
- `_docs_de`

There are different configuration files (one for each language). The configuration files use the collection that pertains to that configuration's language.

Copy the English project folder inside the `_docs` collection and paste it into the Japanese collection (`_docs_ja`) or the German collection (`_docs_de`). This project folder includes both Markdown files and the `images` and `audio` subfolders.

For example, if your project in the `_docs` folder looks like this:

```
├─ _docs
│   └─ myproject
│       ├── images
│       ├── audio
│       ├── file1.md
│       └─ file2.md
```

Your project should look the same in `docs_ja` or `docs_de` :

```
├─ _docs_ja
│   └─ myproject
│       ├── images
│       ├── audio
│       ├── file1.md
│       └─ file2.md
```

Image Paths

You don't need to do anything special to ensure that image paths for translated languages are correctly written. Just make sure that images have the same image files in the `_docs_ja/<project name>/images` folders as you have in `_docs/<project name>/images` .

For example, if you have image `a.png` , `b.png` , and `c.png` in the `_docs/generic/images` version, these same images and file names must appear in the `_docs_ja/generic/images` or `_docs_de/generic/images` folders.

If you decide to localize your screenshots, keep the same file names for the translated files. The best approach to localizing the screenshots may be to have your localization manager get equivalent screenshots using language builds available to them. (Getting a Fire TV in

Japanese, for example, may only be available if you order the device from a Japan location. Work with your localization manager and teach him or her how to capture the screenshots needed.)

As a best practice, don't embed text in graphics. If you have callouts in the text, use numbers that are defined in a legend below the image. Here's an example with callouts in a [Fire App Builder doc page](#).

The `image.html` include supplies the base path, which is populated by a variable in the language's config file. The base path used for the Hippo output varies by language:

- English: `https://images-na.ssl-images-amazon.com/images/G/01/mobile-apps/dex/`
- Japanese: `https://images-na.ssl-images-amazon.com/images/G/01/mobile-apps/dex/ja`
- German: `https://images-na.ssl-images-amazon.com/images/G/01/mobile-apps/dex/de`

Upload your translated images into Media Central. For example, if your project is called `generic`, you would navigate to Media Central, go to `mobile-apps/dex/ja/generic` for Japanese (`mobile-apps/dex/de/generic` for German), and upload your images. Your image paths in the image include would be the same as in English: `generic/<my file>`.

This allows the same image path in your Markdown files to work across all languages. You don't have to adjust any image paths for your translated content.

Content Includes

If you're reusing content across different files through includes, don't worry about trying to send the include source. The translation memory in WorldServer will automatically match similar content and will not charge for the translation of duplicate material.

Conditional Text

If you have some content that is available only for English, or only for Japanese, etc., you can use conditional tags to vary the output. Each config file has a `language` property (with the values `english`, `japanese`, or `german`) that you can use with `if` statements. For example:

```
{% if site.language == "english" %}  
// do this...  
{% endif %}
```

When you build in Japanese or German, this content won't be included.

Use conditional text when a feature might be available only in a specific locale.

Sidebar

The sidebar data sources for each translated language are separate files (for example, into `generic_ja.yml`).

To create a sidebar for your language:

1. In the `_data` folder, duplicate your English sidebar (e.g., `generic.yml`) and add a language suffix to the duplicate like this: `generic_ja.yml` and `generic_de.yml`.
2. Add the language subpath (`ja` or `de`) into the `hurl` values in the sidebar.

In the translated sidebar files, the file names and permalinks must remain the same as in the English sidebar – those aspects of the file aren't translated.

However, the `hurl` values should include the language subpath after `public`. For example, if the English `hurl` is `https://developer.amazon.com/public/some/path/subsublevel3`, the Japanese path would be `https://developer.amazon.com/public/ja/some/path/subsublevel3`. The German path would be `https://developer.amazon.com/public/de/some/path/subsublevel3`.

3. Each of the translated files will need to refer to the correct sidebar in the front matter of the page. Open up each of the pages in your translation folder (for example, `_docs_ja/<project name>`) and adjust the `sidebar` property in the frontmatter from something like `generic` to `generic_ja`.)

Tip: Do a bulk-find-and-replace operation to make this adjustment. In Atom, press **Cmd+Shift+F**. Define the file/directory pattern and make the replacement. Make sure you limit the directory to your translated docs directory only.

4. Adjust the `related_resources_list` property in your sidebar so that only translated assets are listed in this section. For example, point to the forums that support a specific language only.

Documentation homepage

Note: Even though we're not using the homepage in the Hippo output, you must still do this step so that the linking script will work. The linking script depends on the `homepage_products_ja.yml` or `homepage_products_de.yml` file to get a list of products to iterate through.

The list of documentation on the English homepage gets generated from the `products` property in `homepage_products.yml` file in the `_data` folder. For Japanese, the `homepage_products_ja.yml` file is used. For German, the `homepage_products_de.yml` file is used.

The homepage data files for Japanese and German will be much shorter than the files for English. To configure the homepage:

1. Open up `_data/homepage_products_ja.yml` or `_data/homepage_products_de.yml`.
2. Add your project following the same format as the other files.

For the sidebar, remember to reference the translated sidebar, not the English sidebar. For example, `generic_ja.yml`.

Links

The `links.html` script first gathers a list of sidebar names from the `shortname` property in `homepage_products.yml` file (or from `homepage_products_ja.yml` for the Japanese build, or from `homepage_products_de.yml` for the German build). In order for the `links.html` script to work properly, the `homepage_products` file must list out all the sidebars it should loop through.

If you only translated one doc set, but that doc set contains links to other doc sets that aren't translated, add a reference to the other doc sets in the `homepage_products.yml` (or `homepage_products_ja.yml` or `homepage_products_de.yml`) files like this:

```
- shortname: myproject
- shortname: someotherproject
```

Only the `shortname` property is required by the links script. Otherwise links containing the `ref` values in those projects won't be included in the links script.

Note that English sidebars will supply the English links in the Hippo output, not Japanese or German links. There aren't any special classes that insert icons for link targets that only exist in English.

Top Navigation Bar

Note: You can skip this section because we aren't using this theme as the display output for translated languages. However, I've kept it here for potential future use.

The navigation bar at the top of the page is generated from the `topnav.yml` file in `_data`.

Each collection specifies its own `topnav.yml` file. The German `docs_de` collection uses `topnav_de.yml`, and Japanese uses `topnav_ja.yml`.

This topnav value is set as one of the `defaults` in the configuration file for the language, so you never even notice it. However, each page silently uses the frontmatter `defaults` defined for that collection in the config file.

The topnav contains a link to the page on Github (if you have `github: true` as a frontmatter property on your page) and a link to the forum for the product. The forum link is specified in your sidebar file like this:

```
forum:
  forum_name: Fire TV Forum
  forum_link: https://forums.developer.amazon.com/spaces/43/index.html
```

Keep the forum name brief to allow for text expansion in German or Japanese.

Search

Note: You can skip this section because we aren't using this theme as the display output for translated languages. However, I've kept it here for potential future use.

Search is also set up as its own collection and included only in the appropriate language builds. The English build uses the search files in `_search`, Japanese uses `_search_ja` and German uses `_search_de`.

In English, the search file for Fire TV looks as follows:

```
---
layout: default
toc-style: kramdown
github_button: false
sidebar: firetv
type: search
permalink: search-firetv.html
---
```

In German, it looks like this:

```
---
layout: default
toc-style: kramdown
github_button: false
sidebar: firetv_de
type: search
permalink: search-firetv_de.html
---
```

And Japanese:

```
---
layout: default
toc-style: kramdown
github_button: false
sidebar: firetv_ja
type: search
permalink: search-firetv_ja.html
---
```

Note how the `sidebar` and `permalink` properties in the search contain the language suffix.

Open up `_search` and copy your English search file. Paste it into the `search_ja` or `search_de` folder. Open up the file and customize the frontmatter to include the appropriate language suffixes as shown in the examples above.

UI Strings

Various UI strings in the theme are stored in the config file for that language build. The UI strings are extracted into a `uistrings` property.

Currently for Japanese, the strings have been verified by the localization manager. For German, the strings still need to be translated correctly. You can translate these strings by putting them into a `.txt` file and including them with the list of files to be translated.

Provide a Glossary

Translators will need a glossary of terms to consult as they do translations. Glossaries are especially helpful if you aren't consistent in the way you refer to products or services.

If your content has technical jargon (for example, "APK"), it needs to be defined in the glossary so that translators understand it and don't convert it to more natural language.

Follow Localization Best Practices

In addition to the preceding guidelines, also see the [Localization Best Practices for Source Content](#) from the Translation Services Platform team. Here are some highlights from the best practices:

- Keep verb tenses simple (avoid pluperfect future participle gerunds, etc.).
- Keep sentences short. Limit ideas to one idea per sentence.
- Don't stack nouns into long strings.

Push the Files Through the WorldServer Workflow

After your files are properly prepared, you can push them through the WorldServer workflow.

Onboard with WorldServer through TSP

WorldServer is the tool used to handle the files that you push through the translation process. The [Translation Services Platform \(TSP\)](#) team sets up your account in WorldServer and configures it with the right workflow you want for the translations you're making.

The TSP team also gives you \$50k in credit (called a PO, or purchase order) automatically each year, without requiring you to do anything special. If you exceed the \$50k, you will need to ask for an additional PO, and your department's financial person may need to get involved.

The TSP team handles the setup and configuration of your WorldServer account. TSP refer to this process as "[onboarding](#)." (Note: Onboarding does not involve any training. Also, do not confuse onboarding with waterboarding, even though the two experiences may be similar.) Onboarding can take 1-2 weeks, based on the size of the [SIM queue](#). Onboarding involves submitting a MS Word questionnaire as an attachment to a SIM.

A WorldServer account with translation workflows for Japan and German has already been set up for the DevComm team. Moravia is designated as the translation vendor. However, any vendor that can process HTML can be used. You don't have to restrict your choice of vendors to only those who can support Markdown.

You need to set up a new account only if you're guiding another team through the translation workflow that isn't part of the DevComm stewardship. If this is the case, you can [submit a new onboarding request](#) to the TSP team on behalf of the team you're doing doc translation for. (Don't expect an outside group to move files through the WorldServer workflow.)

The TSP team can use the same workflow as our existing DevComm workflow, but to separate out the billing, they assign a different project name to the workflow.

All translation projects should go through the same workflow set up in WorldServer. The translation memory from one translation project will be retained and applied to future translations of the same content.

Each WorldServer configuration has its own translation memory database. You can import translation files or TM into WorldServer if you're migrating from another vendor, but this may take some time.

For TSP questions, reach out to Patrick Magee (patricm). Paul Kasper is also a person to contact for WorldServer and other localization issues.

Generate the HTML Output and Zip the Files

1. Generate the HTML output for your content by running this command:

Japanese:

```
jekyll serve --config _config_hippo_ja.yml
```

German:

```
jekyll serve --config _config_hippo_de.yml
```

The translations will be built in the `_outputs_ja` or `outputs_de` folders – not in the `_site` folder. The `_site` folder contains only the English builds.

Note that these scripts are just shortcuts that store longer commands.

`hippo_ja.sh` tells Jekyll to use the `_config_ja.yml` configuration file and to put the output in the `_output_ja` folder.

2. In the `_output_ja` or `_output_de` folder, get the HTML output of the files.
3. Look in the **hippomenus** folder and get the HTML version of the sidebar file.
4. Compress these files into a zip file.

Note: If you're on a Mac, to avoid having `DS.Store` and `.thumbs.db` files included in the zip file (which will get marked as failed by WorldServer), use a zip tool such as [FolderWasher](#) to “wash” the zip file of unwanted contents.

Submit Your Files in WorldServer

1. With your zipped files ready to go (as described in the previous step, “[Generate the HTML Output and Zip the Files](#)” (page 60)), log in to WorldServer at <https://worldserver9.amazon.com> using your Amazon credentials.

If you can't log in, you need to have the Translation Platform Team set up an account for you. Reach out to Patrick Magee for help.

2. In WorldServer, click **Assignments > Projects** and then **Create New Project**.
3. Specify a name, the project type (**Appstore Developer Communications Documents - Moravia (English Source)**), and the **Locale**. Leave the due date empty.
4. Upload your zip file (one zip file per project), and click **Copy Files Now***.

When you upload the zip file, Moravia (or the assigned vendor) also receives notification via email indicating that a new project has been uploaded. (This automated email notification is part of the workflow setup that the Translation Services Platform team does when setting up your WorldServer account).

However, you should also let your translation manager and Moravia contact know. For Appstore, the Amazon localization manager for Japanese is Nanaho Nishiyama (nanahon). The Moravia contact for Japanese is Ryo Shibata at RyoS@moravia.com. For German, the contact is Marina Lanas at MarinaLa@moravia.com. Ryo is based in Japan, Marina in Argentina. Let them know that you added a new project.

The localization manager interfaces with Moravia to check and adjust the translation. The localization manager's changes apply to the translation memory source so that future updates will contain the changes.

Moravia will estimate the cost for translation and require your manager to approve the estimate prior to the work. Approvals are routed to your manager for approval. There's a workflow set up between Moravia, Amazon billing, and your manager to approve estimates and pay invoices.

Downloading the Translated Files

When you create the project in WorldServer, the workflow creates translation tasks that Moravia "claims." When Moravia finishes with the translation, they finish the project by "completing the task." (These are the terms used in WorldServer.)

When the translation is finished, you will receive an email saying that you have been "assigned a task." At this point you will be able to see and download the translated files.

Note: Before you download the files, ask your translation manager to QA the translation. Any changes the localization manager makes need to be made in the WorldServer source. If you download the files first and the localization manager later has edits, those edits will need to be applied to the WorldServer source, and then you'll need to re-download the files from WorldServer once again. If the localization manager doesn't input edits into the WorldServer source, the changes will be lost the next time you submit files for translation.

To download the translated files:

1. Log into WorldServer.
2. Go to **Assignments > Projects**.
3. In the Project View drop-down menu, select **projects for clients I am a member of**.
4. Click your project's name.
5. In the View drop-down menu, select **all tasks**.
6. Select the check box at the top of the **Task** column.
7. Move your mouse over **More Options** and select **Asset Options > Download**.
8. Select **Target Assets**, and then click **Download**.
9. Uncompress the Zip file and drill into the actual folder containing files.

Getting the Translated Files Back into Your Project

The translated files should **not** be inserted back into the `_docs_ja` or `_docs_de` folder your project. The translation exists in the HTML output only. The files are ready to be copied and pasted into the appropriate pages in Hippo, with no additional work required by Jekyll or you.

You can store the translated HTML files in a subfolder in your project (to keep them safe in case Hippo munges them), but there's no need to overwrite the English Markdown source you originally copied into `_docs_ja` or `_docs_de` folders. When you have updates to make to your translation, you'll go through the same process described in this document again.

Note: We are relying on the accuracy of translation memory in WorldServer to recognize content that is the same. If you change 100 words across 30 files and resubmit them for re-translation, the cost of the re-translation will be limited to the 100 sentences in which those 100 words appear. You should not be charged for re-translating all the pages that have already been translated and whose source is stored in translation memory in WorldServer.

Note that translation memory in WorldServer is separated out by project. If you change the project name, the translation memory will not be available unless the translation team manually exports it and reimports it into the new project.

Warning: If you're using a new vendor, make sure they are storing the translated content in translation memory and not just translating off the cuff in a Word file.

Publishing on Hippo

1. Upload the translated sidebar menu into the [same place in Media Central](#) where we store all sidebar menus. (Unlike with images, the sidebar menus are all contained in the same Media Central folder, not within subfolders. Subfolders aren't necessary because the sidebar menu names are all unique.)
2. To publish the page content, in the [Hippo console](#), browse to the English version of the page.
3. Click the English language menu and then select **Japanese**. If the Japanese page does not exist, a green plus sign will appear next to Japanese. Clicking this button will clone the English page into the corresponding directory in the Japanese section of the site.
4. Use the same file names and directory paths for Japanese content as you do for English.
5. Paste in the Japanese or German content into the cloned Hippo page and publish

as usual.

Note: Hippo does not support German. As a result, if you need to publish German, reach out to Aperture with questions.

Publishing Updates

When you have updates to your content, you don't need to meticulously keep track of all changed files or send diff files to TSP. Instead, just zip your files up and resubmit them through the same process as before. The translation memory in WorldServer will match on previously translated strings. You will only be charged for new or updated strings that need new translation.

Troubleshooting

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

Issues building the site

Address already in use

When you try to build the site, you get this error in iTerm:

```
jeekyll 2.5.3 | Error: Address already in use - bind(2)
```

This happens if a server is already in use. To fix this, edit your config file and change the port to a unique number.

If the previous server wasn't shut down properly, you can kill the server process using these commands:

```
ps aux | grep jeekyll
```

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

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

Alternatively, type the following to stop all Jekyll servers:

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

shell file not executable

If you run into permissions errors trying to run a shell script file (such as `jeekyllhowto-multibuild_web.sh`), you may need to change the file permissions to make the sh file executable. Browse to the directory containing the shell script and run the following:

```
chmod +x build_writer.sh
```

shell file not runnable

If you're using a PC, rename your shell files with a `.bat` extension.

"page 0" cross references in the PDF

If you see "page 0" cross-references in the PDF, the URL doesn't exist. Check to make sure you actually included this page in the build.

If it's not a page but rather a file, you need to add a `noCrossRef` class to the file so that your print stylesheet excludes the counter from it. Add `class="noCrossRef"` as an attribute to the link.

The PDF is blank

Check the `prince-list.txt` file in the output to see if it contains links. If not, you have something wrong with the logic in the `prince-list.txt` file. Check the `conditions.html` file in your `_includes` to see if the audience specified in your configuration file aligns with the `buildAudience` in the `conditions.html` file

Sidebar not appearing

If you build your site but the sidebar doesn't appear, check that your page has a `sidebar` property in the frontmatter that corresponds with a sidebar in the `_data` folder.

Sidebar isn't collapsed

If the sidebar levels aren't collapsed, usually your JavaScript is broken somewhere. Open the JavaScript Console and look to see where the problem is. If one script breaks, then other scripts will break too, so troubleshooting it is a little tricky.

Search isn't working

If the search isn't working, check the JSON validity in the `search.json` file in your output folder. Usually something is invalid. Identify the problematic line, fix the file, or put `search: exclude` in the frontmatter of the file to exclude it from search.

Links not working

Make sure you add `{% include links.html %}` at the bottom of each page. Make sure your sidebar is listed in the `sidebars` property in your configuration file.

Links include isn't working correctly

If you look at your HTML output and see that the `links.html` include at the bottom is leaving all the Markdown reference tags in the output, something is wrong with your Markdown or HTML syntax on the page. As a result, the Markdown might stop rendering.

For example, look at your HTML tables. If there is one tiny formatting issue (a tag not properly closed), kramdown will stop working on the file and won't convert any other content in the file. The Markdown reference tags are just evidence that the Markdown filter stopped converting the file to HTML at this point. Fix the HTML and the Markdown will properly convert to HTML.

Building Someone Else's Project

When you build someone else's project, if the person has a Gem and Gemfile, you might get errors like this:

```
WARN: Unresolved specs during Gem::Specification.reset:
      rouge (~> 1.7)
      jekyll-watch (~> 1.1)
WARN: Clearing out unresolved specs.
```

This is because the person had different version of Ruby gems on their computer, which get packaged into a Gemfile.

First run bundler to make sure the gem versions on your computer match those of the project. From the project's directory, run the following:

```
bundle install
```

Then run this:

```
bundle exec jekyll serve --config configs/jekyll_english.yml
```

Now you can press **Ctrl+C** and run `. jekyll.sh` as usual.

Checking the version of Jekyll

You can see what version of Jekyll you have by running the following:

```
jekyll -version
```

General Jekyll Topics

In this section:

About Ruby Gems and Bundler.....	68
Install Jekyll on Mac	74
Install Jekyll on Windows	78
Tips for Atom Text Editor.....	81

About Ruby, Gems, Bundler, and other prerequisites

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 \(page 68\)](#)
- [About Ruby Gems \(page 68\)](#)
- [Rubygem package managers \(page 68\)](#)
- [Gemfiles \(page 69\)](#)
- [Gemfile.lock \(page 70\)](#)

About Ruby

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

About Ruby Gems

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

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

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

Rubygem package managers

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

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

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

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

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

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

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

Gemfiles

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

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

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

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

The gems it retrieves are listed separately on each line.


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

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

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

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

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

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

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

If it adds another digit, the scope is affected:

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

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

If it looks like this:

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

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

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

Gemfile.lock

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

GEM

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

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



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

PLATFORMS

ruby

DEPENDENCIES

github-pages
jekyll

BUNDLED WITH

1.11.2

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

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

Install Jekyll on Mac

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 \(page 74\)](#)
- [Install Homebrew \(page 75\)](#)
- [Install Ruby through Homebrew \(page 75\)](#)
- [Installing dependencies through Bundler \(page 76\)](#)
- [Serve the Jekyll Documentation theme \(page 76\)](#)
- [Resolve “No Github API authentication” errors \(page 76\)](#)

Ruby and RubyGems

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

If you get responses that look like this:

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

and

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

Great! Skip down to the [Bundler \(page 75\)](#) section.

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

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

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

Install Homebrew

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

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

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

```
brew update
```

Install Ruby through Homebrew

Now use Homebrew to install Ruby:

```
brew install ruby
```

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

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

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

And this:

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

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

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

Install the Jekyll gem

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

Now use `gem` to install Jekyll:

```
gem install jekyll
```

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

Installing dependencies through Bundler

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

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

You install Bundler by using the gem command with RubyGems:

```
gem install bundler
```

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

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

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

Serve the Jekyll Documentation theme

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

Resolve “No Github API authentication” errors

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

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

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

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

To resolve the “No Github API authentication” error:

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

```
open ~/.bash_profile
```

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

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

```
export JEKYLL_GITHUB_TOKEN=abc123abc123abc123abc123abc123abc123abc123abc123
```

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

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

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

6. Close and restart your terminal.

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

Install Jekyll on Windows

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

Install Ruby

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

1. Go to [RubyInstaller for Windows](#).
2. Under **RubyInstallers**, download and install one of the Ruby installers (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](#).
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](#).

Install the Jekyll gem

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

Now use `gem` to install Jekyll:

```
gem install jekyll
```

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

Installing dependencies through Bundler

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

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

You install Bundler by using the gem command with RubyGems:

Install Bundler

1. 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](#) 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](#) instead. The Git Bash client will allow you to run shell scripts and execute other Unix commands.

Serve the Jekyll Documentation theme

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

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

Resolving Github Metadata errors

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

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

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

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

Atom Text Editor

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.

- [Atom Shortcuts \(page 81\)](#)

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

Customize the invisibles and tab spacing in Atom:

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

Turn off auto-complete:

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

Atom Shortcuts

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

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