Jekyll How-to Guide

Version 1.0

Last generated: July 04, 2017

Table of Contents

Get Started

In this section:

Jekyll Doc Project: A Jekyll theme for documentation

The Jekyll Doc Project theme is intended for technical documentation projects, such as user guides for software, hardware, and APIs. This Jekyll theme (which has an Apache 2.0 open source license) uses pages exclusively and features robust multi-level navigation. It includes both web and PDF output. The GitHub repo is here: amzn/jekyll-theme-doc-project.

Installation

Before you can install the project, you may need to install Jekyll.

1. Install Jekyll:
    - Install Jekyll on Mac (page 59)
    - Install Jekyll on Windows (page 62)

    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. Download the theme from the amzn/jekyll-theme-doc-project repo.
3. Configure the values in the **_config.yml** file based on the inline code comments in that file.
4. Serve or build the Jekyll site:

    ```
    jekyll serve
    ```

Overwriting Theme Files

You'll most likely want to add some of your own styles, esp. to style the top navigation bar and other elements. There are three files specifically set up for this:

- **assets/css/pdf/user_defined_web_styles.css** (web styles)
- **assets/css/pdf/user_defined_pdf_styles.css** (print styles)
- **assets/js/user_defined_javascript.js** (web javascript)

These files are blank but are referenced in the theme's layout files. By adding these files in your project and defining your styles, they will automatically be included in the layout.

Content Types

This theme uses collections exclusively (instead of posts). Collections are a content type, like a page or post. In this case, the content type is a document within the "docs" folder. Documents function just like pages, for the most part. (When referring to them in this documentation, "pages" and "documents" are used interchangeably.) Organize your pages inside the **_docs** folder.

Organize your pages with the folder structure that you want displayed in the breadcrumb (page 7). Use the spacing and capitalization in your folder names that you want reflected in your breadcrumb path. For example, "Getting Started" instead of "getting_started".

The folder structure informs the breadcrumb display only. Upon build, the site output will pull all of the pages out of their folders, subfolders, etc., and put them into the root directory of _site. This "flattening" of the page hierarchy enables relative linking in the project. 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.

Breadcrumbs

If you want breadcrumbs in your project, set these properties in the _config.yml file:

```
# Display breadcrumbs at all?
breadcrumb_display: true

# Display Home path in breadcrumb?
breadcrumb_home_display: true

# Url for Home path to point to
breadcrumb_home_url: http://yourcompanysite.com

# Display name for "Home" path
breadcrumb_home_name: Home
```

Jekyll projects require an index.html (or index.md) file in your root directory. This is the page that loads by default. Because this page lives outside your regular _docs directory, you have to manually define the breadcrumb path (after Home) that you want there.

Note that clicking the folder paths in the breadcrumb doesn't do anything except for Home. Visually, all paths look the same, but no logic is configured to dynamically render a list of items contained in file path folders.

Images

You can store your images either inside or outside your project. In both cases, use the image include (page 37) to insert images in your pages.

Storing Images Inside the Project

If you want to store your images inside your Jekyll project, store the images in the _docs/images folder. In the _config.yml file, set the image path that will be prepended to the image file name. By default, the output will show the images in the same folder structure as you have in your Jekyll project source directory. If you put your images in **\/images**, then when your Jekyll site builds, your images will be in **\/images**. Rather than writing **images/myfile.png** in the `file` parameter of the image include, you can put the image path in the config file's `image_path` parameter:

```
image_path: images
```

This path will be prepended before image file names. (Note that a trailing slash is added automatically in the image include.)

This way, when you use the image include to insert images, you can simply write **myfile.png** for the `file` parameter.

Storing Images Outside Your Project

Sometimes you might want to store your images outside your Jekyll build. Git repos don't handle large numbers of images well (your .git files become huge), since images are binaries. If you want to store images outside your project, open **.gitignore** and add **images** to it.

In the `image_path` property in the _config.yml file, define the path where the images will be available:

```
image_path: https://s3-us-west-1.amazonaws.com/my-bucket-name/my-proje
ct-images
```

If you're using AWS S3, you can transfer images to your server from the command line. See AWS Command Line Interface for details. After installing the aws cli, you'll need to configure your credentials and determine the right commands to transfer files to your bucket.

Within your Jekyll project, you could add a shell script that contains the command needed to transfer files to your bucket, like this:

```
aws s3 sync images s3://my-bucket-name/my-project-images
```

The images_upload.sh contains this generic AWS CLI command. The S3 bucket will store the images in a structure like this:

```
my-bucket-name > my-project-images > image1.png

├── my-bucket-name
│   ├── my-project-images
│      ├── image1.png
```

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
---
```

(Even if it's a Markdown file, the `permalink` property should specify the .html extension, since this is how the file will appear in the output.)

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.
permalink	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.
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 could be used in integrations with more robust search services.

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

```
defaults:

  -
    scope:
      path: ""
      type: docs/myproject
    values:
      product: My Project
      sidebar: myprojectsidebar
```

In this example, all documents in the _docs/myproject file will automatically have `product: My Project` and `sidebar: myprojectsidebar` as values in the frontmatter when the site builds.

Sidebar Configuration

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. See Sidebar Navigation (page 12) for more details.

Top Navigation Bar

You configure the top navigation bar through the _data/topnav.yml file. You can configure single links or links with drop downs. If you want the search to appear in the sidebar rather than the top nav, see these options in the _config.yml file:

```
search_in_topnav: true
search_in_sidebar: false
```

Placing the search in the top navigation bar gives the theme more visual balance.

When you shrink the browser size, the navbar options change to a couple of icons. You can control whether those icons appear with these settings in _config.yml:

```
toggle-sidebar-button: none
navbar-toggle: none
```

(You might want to hide these buttons if your nav bar doesn't have any links or options, or if the sidebar is empty.)

Configure the Footer

You configure the footer through the options in _data/footer.yml.

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 generic.yml or jekyllhowto.yml files. The syntax for your YAML content must be correct in order for the files to be valid.

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.

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, hard-coded URL, and the Markdown referent for linking. Here's an example:

```
  - title: Sample1
    jurl: /sample.html
    hurl: /solutions/widgets/acme/docs/sample
    ref: sample1
```

Note: The `hurl` value is optional. This is used only in systems where the relative URL may not work. Some server environments require the absolute, hard-coded URL to the page. If so, you can use this `hurl` value. (Unless you have an odd publishing system, you probably won't use `hurl` and can omit it.)

Property	Description
`title`	The page title.
`hurl`	The hard-coded (absolute) URL to your content. (This is only required if you're publishing to a system that requires hard-coded URLs. Otherwise, omit this.)

Property	Description
`jurl`	The Jekyll URL. Use a relative link that begins with a `/` 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).
`ref`	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.

The `ref` property is like a variable used to refer to either the `hurl` or `jurl` 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
```

The sidebar will show this information below the sidebar.

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://docs.mycompany.com/index
    ref: home

  - title: Getting Started
    jurl: /getting-started.html
    hurl: https://docs.mycompany.com/getting-started.
    ref: home

- title: Sample Folder Title
  folderitems:

  - title: Sample1 level1
    jurl: /sample.html
    hurl: https://docs.mycompany.com/solutions/widgets/acme/docs/sampl
e
    ref: sample1

  - title: Sample2 level1
    jurl: /sample2.html
    hurl: https://docs.mycompany.com/solutions/widgets/acme/docs/sampl
e2
    ref: sample12

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

      - title: Some content
        jurl: /sublevel1.html
        hurl: https://docs.mycompany.com/solutions/widgets/acme/docs/s
ublevel1
        ref: sublevel1

      - title: Some more content
        jurl: /sublevel2.html
        hurl: https://docs.mycompany.com/solutions/widgets/acme/docs/s
ublevel2
        ref: sublevel2

        subsubfolders:
```

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

          - title: Some content last level
            jurl: /subsublevel1.html
            hurl: https://docs.mycompany.com/solutions/widgets/acme/do
cs/subsublevel1
            ref: sample

          - title: Some more content last level
            jurl: /subsublevel2.html
            hurl: https://docs.mycompany.com/solutions/widgets/acme/do
cs/subsublevel2
            ref: sample
```

As you can see, there's a `folders` object that 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 general 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.

Level Requirements

Note that you must have items at each level. If you want to have a folder that contains 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 1
      output: web
      subfolderitems:

      - title: Some content
        jurl: /sublevela.html
        hurl: https://docs.mycompany.com/i/solutions/widgets/acme/doc
s/sublevela
        ref: sublevela

    - title: My child folder 2
      output: web
      subfolderitems:

      - title: Some content
        jurl: /sublevelb.html
        hurl: https://docs.mycompany.com/i/solutions/widgets/acme/doc
s/sublevelb
        ref: sublevelb
```

This structure looks like this:

```
├── My parent folder
│   ├── My child folder 1
│      └── sublevela.html
│   ├── My child folder 2
│      └── sublevelb.html
```

☑ **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.

Customize the Header

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.

- Tooltips (page 41)
- Navtabs (page 42)
- Workflow Maps (page 45)

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:

```

```

If you don't have `` 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

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

   Here’s a list in here:

   - first item
   - second item

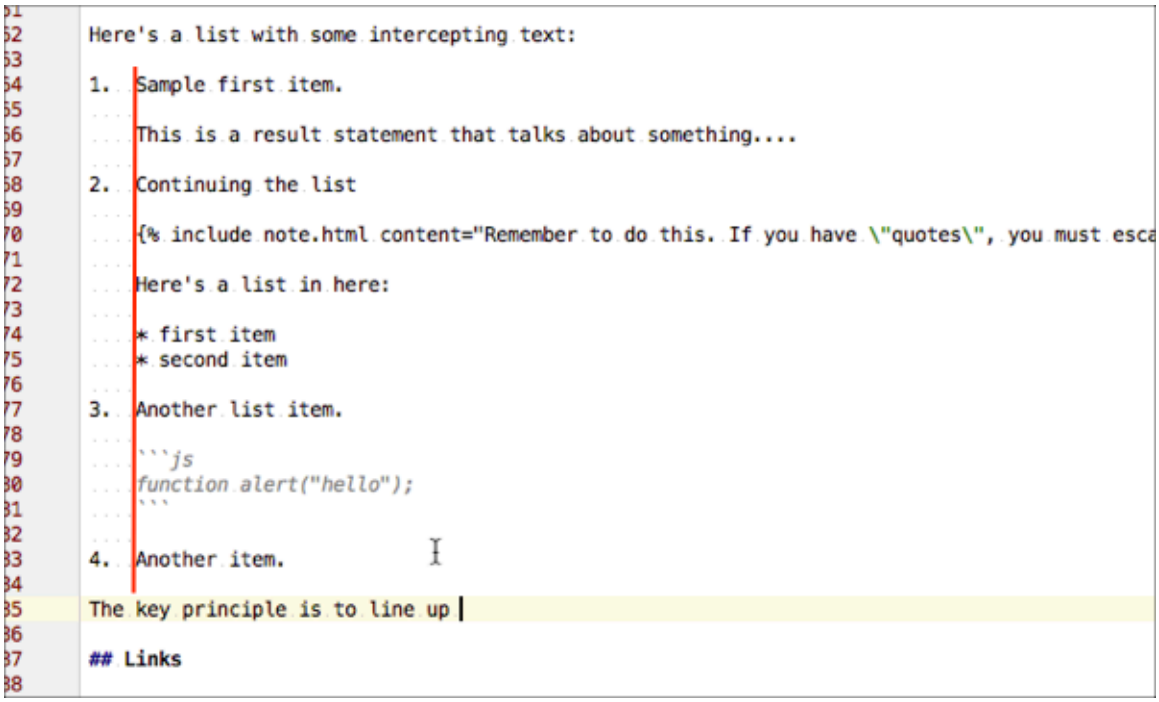
3. Another list item.

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

4. Another item.

Key Principle to Remember with Lists

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



```
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="Remember to do this. If you have \"quotes\", you must esca

    Here's a list in here:

    * first item
    * second item

3.  Another list item.

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

4. Another item.

The key principle is to line up

Links
```

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:

> ℹ **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
`content`	The content for the parameter.	Required
`type`	The color for the callout. Options are info, warning, danger, success, primary, default.	Required
`title`	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 annou
nce
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 t
o 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

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 %}
```

> ℹ **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.ht
ml).
```

Result:

See the Android documentation.

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][ColorOverlayD
immer] class in the Android documentation.

...

[cfdim]: https://developer.android.com/reference/android/service/vr/Vr
ListenerService.html
[codim]: https://developer.android.com/reference/android/support/v17/l
eanback/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 _date 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="jekyllh
owto/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 `<span class="red">` 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 `<span>` 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 apples	Second priority	Third 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 `<br/><br/>`.

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 Ub
er, 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…	**Use this skill type**
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	Custom skill (*custom interaction model*) See Sample 3 (page 0) …(more content…)
…(content in second row, first column)	…(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="company_logo.png" url="http://developer.co
mpany.com" alt="My alternative image text" caption="This is my captio
n" border="true" max-width="90%" %}
```

Result:

This is my caption
The image include's properties are as follows:

Property	Description	Required?
`file`	The name of the file (include the file extension)	Required

Property	Description	Required?
`url`	Whether to link the image to a URL	Optional
`alt`	Alternative image text for accessibility and SEO	Optional
`caption`	A caption for the image	Optional
`border`	A border around the image. If you want the border, set this equal to `true`. Otherwise omit the parameter.	Optional
`max-width`	You can use px or a percentage, such as `70px`.	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="company_logo.png" url="http://dev.compan
y.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="androidsdkmanagericon.png" alt="SDK button" border="true" max-wi
dth="90%" %}
```

Result:

Click the **Android SDK Manager** button 

The inline_image.html include properties are as follows:

Property	Description	Required
`file`	The name of the file (include the file extension)	Required
`alt`	Alternative image text for accessibility and SEO	Optional
`border`	A border around the image. If you want the border, set this equal to `true`. Otherwise omit the parameter.	Optional
`max-width`	A maximum width for the image. You can use px or a percentage, such as `70px`.	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 molecu
les 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 molecu
les 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, riperian]
```

Then make links in your content like this:

Result:

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

Tooltips

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

```
The setting was {% include tooltips.html term="riperian" capitalize="t
rue" %} 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, `<a href='https://en.wikipedia.org/wiki/River'>River in Wikipedia</a>`.
- 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:

Parameters	Description
term	The glossary term in the glossary.yml file
capitalize	Include only if you want the term capitalized. If so, set it equal to `true` 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="noExtIcon" href="#firsttab" data-toggl
e="tab">First Tab</a></li>
  <li><a class="noExtIcon" href="#secondtab" data-toggle="tab">Second
Tab</a></li>
  <li><a class="noExtIcon" href="#thirdtab" data-toggle="tab">Third Ta
b</a></li>
  <li><a class="noExtIcon" href="#fourthtab" data-toggle="tab">Fourth
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
```

```
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="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.6b68exxxxxxxxxx9",
        "notificationTime": "2016-12-02T21:09:58.689Z",
        "notificationId": "amzn1.dash.notification.v1.xxxxxxxxxxxxx13",
        "version": "2015-06-05"
    }
```

</p>
```

Workflow Maps

You can include a simple linear workflow map that has squares at the top depicting a process. Add the following in your page's frontmatter:

```
simple_map: true
map_name: <filename.html>
```

Inside the _includes folder (create one if you don't have one), create the referenced map_name file and follow this format:

```
<div id="userMap">
<div class="content"><a href="file-1.html"><div class="box box1"><spa
n class="stepName">STEP 1:</span> <br/>Download the Project</di
v></a></div>
<div class="arrow">→</div>
<div class="content"><a href="file-2.html"><div class="box box2"><spa
n class="stepName">STEP 2:</span><br/>Configure the Settings</di
v></a></div>
<div class="arrow">→</div>
<div class="content"><a href="file-3.html"><div class="box box3"><spa
n class="stepName">STEP 3:</span><br/>Run the Installer</div></a></di
v>
</div>
```

Customize the text and links that you want. Note that you can't use automated links here. You have to hard-code the links in HTML.

Also note that you can't have too many workflow squares (the max is 5-6).

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

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.

> ℹ **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.

9. 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.

10. 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:

```
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 47).
2. Build the HTML-friendly version of the site:

   ```
   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 p
   df/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 52).
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`.

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:

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

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

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

`ps aux | grep jekyll`

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

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

Alternatively, type the following to stop all Jekyll servers:

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

shell file not executable

If you run into permissions errors trying to run a shell script file (such as jekyllhowto-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 `noExtIcon` class to the file so that your print stylesheet excludes the counter from it. Add `class="noExtIcon"` 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:

```
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:

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

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 60) 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 di
rectory.
```

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 user-specific paths, try restarting your computer. If you're still having trouble, try making these directories writeable. If that still doesn't work, 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
```

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.

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.

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.)

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.

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**.

Adjust the display for auto-complete

If you don't adjust the timing on the auto-complete, the constant prompts from the editor get annoying. At the same time, when you want the prompts, it's extremely helpful to have this feature.

1. Go to **Atom > Preferences**.
2. Click the **Packages** tab.
3. Search for **autocomplete-plus**.
4. In the **Display Before Suggestions Shown** box, type **300**.
5. In the **Keymap for Confirming a Suggestion** box, select **tab and enter**.

Show files listed in .gitignore

By default, files listed in .gitignore will be hidden from view in Atom. You can adjust this by doing the following:

1. Go to **Atom > Preferences**.
2. Click the **Packages** tab.
3. Search for **Tree View** package.
4. Click **Settings**.
5. Clear the check box that says **Hide VCS Ignored Files**.

Atom Shortcuts

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

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

Atom Snippets

You can use a number of shortcuts with Atom. These shortcuts are entered as snippets. You have to enter these shortcuts manually in your Atom editor.

Use Snippets

Snippets expand stored text from a keyword shortcut. For example, if you type **ximage** and then press **Enter**, it expands to {% include image.html file=”” max-width=”” border=”” url=”” caption=”” alt=”” %} assuming you have this snippet configured.

To configure snippets:

1. Copy the snippets file in **assets/snippets.txt**.
2. In Atom, go to **Atom > Snippets**.
3. Replace the existing snippet contents with the content you pasted. Make sure your cursor is flush left before pasting the content.

Here are the shortcuts:

- xtoc
- xfront
- xcallout
- xnote
- xtip
- xcaution
- xwarning
- xlinks
- ximage
- xinline_image
- xaudio
- xbookmark

- xcomment
- xcode
- xnavtabs
- xtable

To insert the code from the snippet, type the shortcut and press **Enter**. Note that you have to be in a Markdown file for the snippets to work. The snippets are associated with Markdown files.

If you have the “autocomplete plus” package turned on in Atom, you’ll see prompts when you enter this text. You can also add your own shortcuts to your snippets file by following the same format.