



Foliant

User's Manual

Welcome to Foliant!

Foliant is a all-in-one documentation authoring tool. It lets you produce standalone documents in pdf and docx, as well as websites, from single Markdown source.

Foliant is a higher order tool, which means it uses other programs to do its job. For pdf and docx, it uses [Pandoc](#), for websites it uses [MkDocs](#).

Foliant preprocessors let you include parts of documents in other documents, show and hide content with flags, render diagrams from text, and much more.

Logo made by [Hand Drawn Goods](#) from [www.flaticon.com](#).

Installation

Installing Foliant to your system can be split into three stages: installing Python with your system's package manager, installing Foliant with pip, and optionally installing Pandoc and TeXLive bundle. Below you'll find the instructions for three popular platforms: macOS, Windows, and Ubuntu.

Alternatively, you can avoid installing Foliant and its dependencies on your system by using Docker and Docker Compose.

macOS

1. Install Python 3.6 with Homebrew:

```
$ brew install python3
```

2. Install Foliant with pip:

```
$ python3 -m pip install foliant foliantcontrib.init
```

3. If you plan to bake pdf or docx, install Pandoc and MacTeX with Homebrew:

```
$ brew install pandoc mactex librsvg
```

Windows

0. Install [Scoop package manager](#) in PowerShell:

```
$ iex (new-object net.webclient).downloadstring('https://get.scoop.sh')
```

1. Install Python 3.6 with Scoop:

```
$ scoop install python
```

2. Install Foliant with pip:

```
$ python -m pip install foliant foliantcontrib.init
```

3. If you plan to bake pdf or docx, install Pandoc and MikTeX with Scoop:

```
$ scoop install pandoc latex
```

Ubuntu

1. Install Python 3.6 with apt. On 14.04 and 16.04:

```
1 $ add-apt-repository ppa:jonathonf/python-3.6
2 $ apt update && apt install -y python3 python3-pip
```

On newer versions:

```
$ apt update && apt install -y python3 python3-pip
```

2. Install Foliant with pip:

```
$ python3.6 -m pip install foliant foliantcontrib.init
```

3. If you plan to bake pdf or docx, install Pandoc and TeXLive with apt and wget:

```
1 $ apt update && apt install -y wget texlive-full librsvg2
  -bin
2 $ wget https://github.com/jgm/pandoc/releases/download
  /2.0.5/pandoc-2.0.5-1-amd64.deb && dpkg -i pandoc
  -2.0.5-1-amd64.deb
```

Docker

If you use `foliant init`, `Dockerfile` and `docker-compose.yml` files to build the project are created automatically. To build the project, run:

```
1 # Site:
2 $ docker-compose run --rm <project-name> make site
3 # Pdf:
4 $ docker-compose run --rm <project-name> make pdf
```

Alternatively, you can create the Dockerfile manually using Foliant's official Docker images:

```
1 FROM foliant/foliant
2 # If you plan to bake PDFs, uncomment this line and comment
  the line above:
3 # FROM foliant/foliant:pandoc
4 RUN pip3 install foliantcontrib.mkdocs
```

Then, run Foliant in a container:

```
1 # Site:
2 $ docker run --rm -it -v `pwd`: /usr/src/app -w /usr/src/app
  my-project make site
3 # Pdf:
4 $ docker run --rm -it -v `pwd`: /usr/src/app -w /usr/src/app
  my-project make pdf
```

Quickstart

In this tutorial, you'll learn how to use Foliant to build websites and pdf documents from a single Markdown source. You'll also learn how to use Foliant preprocessors.

Create New Project

All Foliant projects must adhere to a certain structure. Luckily, you don't have to memorize it thanks to Init extension.

You should have installed it during Foliant installation and it's included in Foliant's default Docker image.

```
1 $ foliant init
2 Enter the project name: Hello Foliant✓
3 Generating Foliant project—————
4
5 Project "Hello Foliant" created in hello-foliant
6 $ cd hello-foliant
7 $ tree
8 .├──
9   foliant.yml└──
10  src
11    └── index.md
12
13 1 directory, 2 files
```

To do the same with Docker, run:

```
1 $ docker run --rm -it -v `pwd`: /usr/app/src -w /usr/app/src
   foliant/foliant init
2 Enter the project name: Hello Foliant✓
3 Generating Foliant project—————
4
5 Project "Hello Foliant" created in hello-foliant
```

Init command created a config file `foliant.yml` and a source directory `src` with one source file `index.md`. And that is the simplest Foliant project!

Build Site

In the project directory, run:

```
1 $ foliant make site✓  
2 Parsing config✓  
3 Applying preprocessor mkdocs✓  
4 Making site with MkDocs—————  
5  
6 Result: Hello_Foliant-2018-01-23.mkdocs
```

Or, with Docker Compose:

```
1 $ docker-compose run --rm hello-foliant make site✓  
2 Parsing config✓  
3 Applying preprocessor mkdocs✓  
4 Making site with MkDocs—————  
5  
6 Result: Hello_Foliant-2018-01-23.mkdocs
```

That's it! Your static, MkDocs-powered website is ready. To view it, use any web server, for example, Python's built-in one:

```
1 $ cd Hello_Foliant-2018-01-23.mkdocs  
2 $ python -m http.server  
3 Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Open localhost:8000 in your web browser. You should see something like this:



Figure 1. Basic Foliant project built with MkDocs

Build Pdf

Note

To build pdfs with Pandoc, make sure you have it and TeXLive installed (see Installation).

In the project directory, run:

```
1 $ foliant make pdf✓
2 Parsing config✓
3 Applying preprocessor flatten✓
4 Making pdf with Pandoc—————
5
6 Result: Hello_Foliant-2018-01-23.pdf
```

To build pdf in Docker container, first uncomment `foliant/foliant:pandoc` in your project's `Dockerfile`:

```
1 - FROM foliant/foliant
2 + # FROM foliant/foliant
```



```
3 # If you plan to bake PDFs, uncomment this line and comment
  the line above:
4 - # FROM foliant/foliant:pandoc
5 + FROM foliant/foliant:pandoc
6
7 COPY requirements.txt .
8
9 RUN pip3 install -r requirements.txt
```

Note

Run `docker-compose build` to rebuild the image from the new base image if you have previously run `docker-compose run` with the old one.

Then, run this command in the project directory:

```
1 $ docker-compose run --rm hello-foliant make pdf✓
2 Parsing config✓
3 Applying preprocessor flatten✓
4 Making pdf with Pandoc—————
5
6 Result: Hello_Foliant-2018-01-23.pdf
```

Your standalone pdf documentation is ready! It should look something like this:

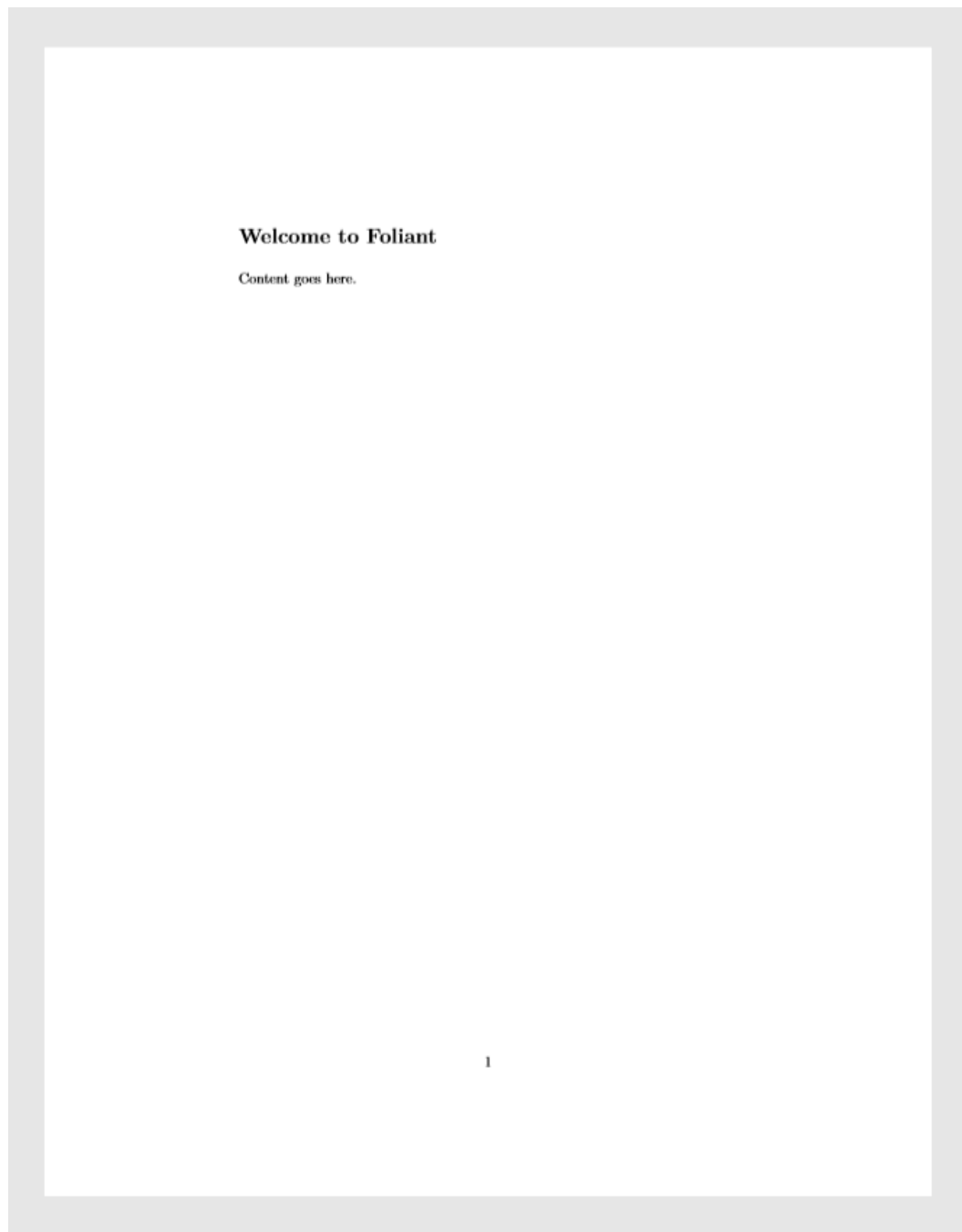


Figure 2. Basic Foliant project built with Pandoc

Edit Content

Your project's content lives in *.md files in src folder. You can split it between multiple files and subfolders.

Foliant encourages [pure Markdown](#) syntax as described by John Gruber. Pandoc, MkDocs, and other backend-specific additions are allowed, but we strongly recommend to put them in `<if></if>` blocks.

Create a file `hello.md` in `src` with the following content:

```
1 # Hello Again
2
3 This is regular text generated from regular Markdown.
4
5 Foliant doesn't force any *special* Markdown flavor.
```

Open `foliant.yml` and add `hello.md` to `chapters`:

```
1 title: Hello Foliant
2
3 chapters:
4   - index.md
5 + - hello.md
```

Rebuild the project to see the new page:

```
1 $ docker-compose run --rm hello-foliant make site && docker-
  compose run --rm hello-foliant make pdf✓
2 Parsing config✓
3 Applying preprocessor mkdocs✓
4 Making site with MkDocs—————
5
6 Result: Hello_Foliant-2018-02-08.mkdocs✓
7 Parsing config✓
8 Applying preprocessor flatten✓
9 Making pdf with Pandoc—————
10
11 Result: Hello_Foliant-2018-02-08.pdf
```

And see the new page appear on the site and in the pdf document:

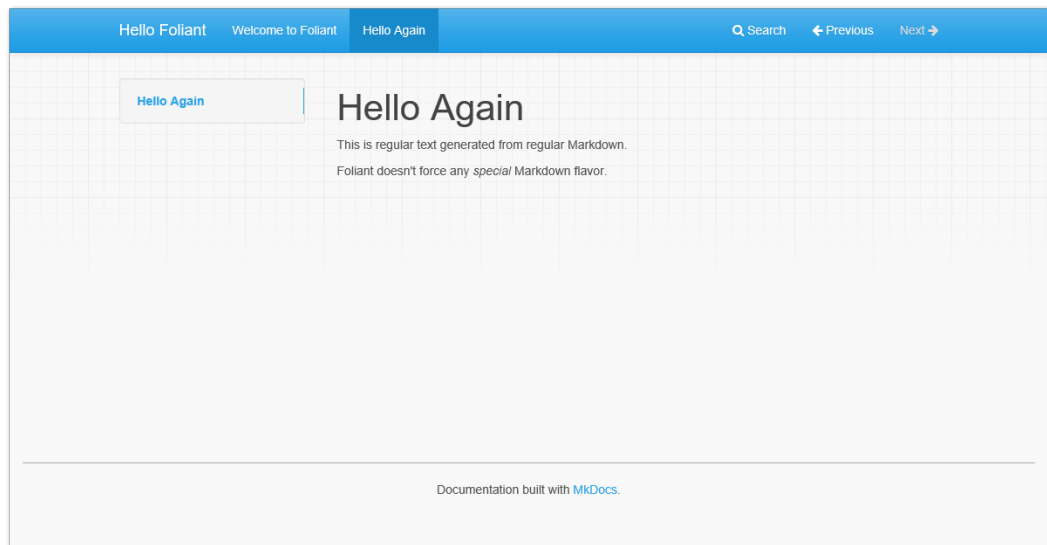


Figure 3. New page on the site

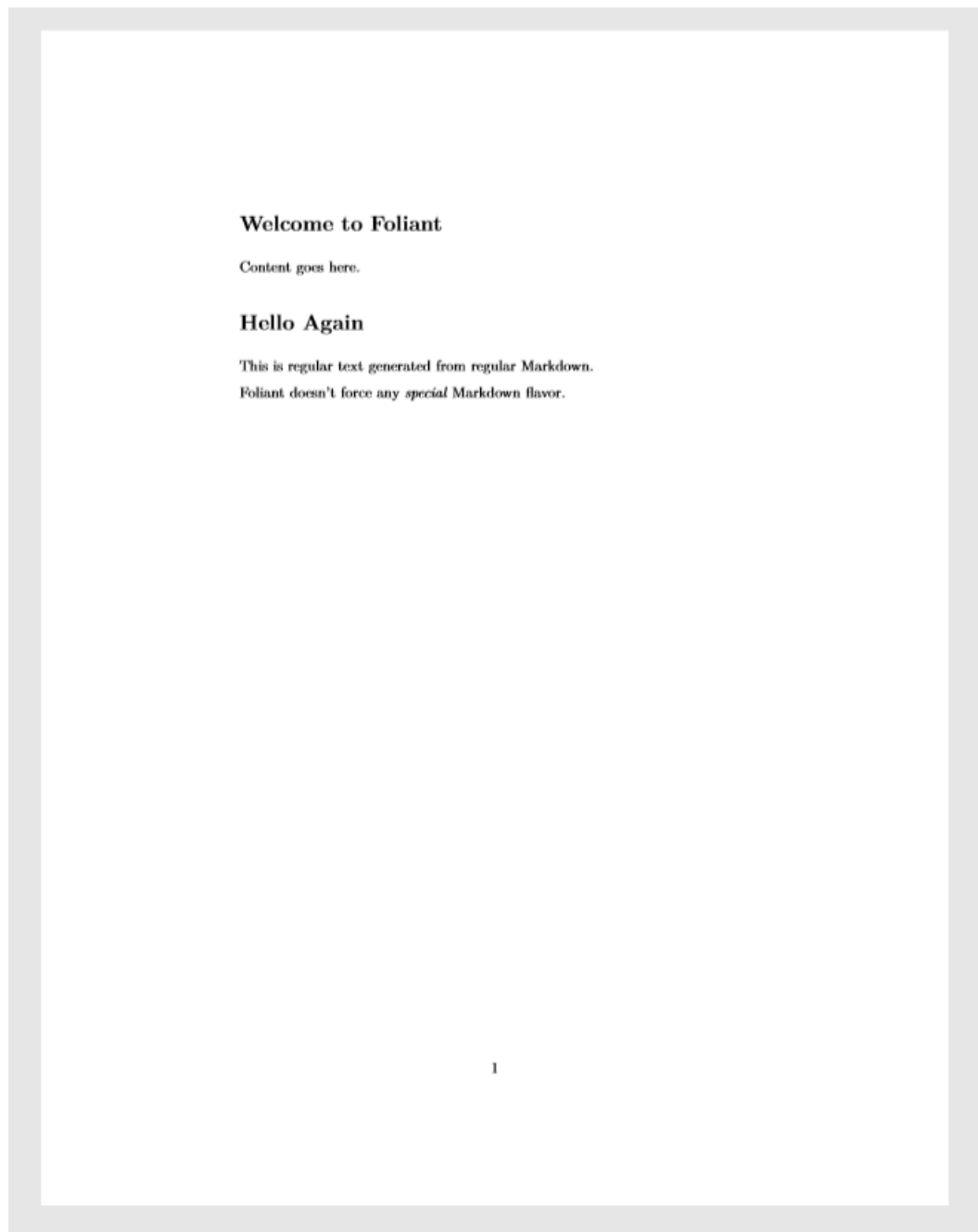


Figure 4. New page in the pdf document

Use Preprocessors

TODO

Backends

MkDocs

MkDocs backend lets you build websites from Foliant projects using [MkDocs](#) static site generator.

The backend adds three targets: `mkdocs`, `site`, and `ghp`. The first one converts a Foliant project into a MkDocs project without building any html files. The second one builds a standalone website. The last one deploys the website to GitHub Pages.

Installation

```
$ pip install foliantcontrib.mkdocs
```

Usage

Convert Foliant project to MkDocs:

```
1 $ foliant make mkdocs -p my-project✓
2 Parsing config✓
3 Applying preprocessor mkdocs✓
4 Making mkdocs with MkDocs—————
5
6 Result: My_Project-2017-12-04.mkdocs.src
```

Build a standalone website:

```
1 $ foliant make site -p my-project✓
2 Parsing config✓
3 Applying preprocessor mkdocs✓
4 Making site with MkDocs—————
5
6 Result: My_Project-2017-12-04.mkdocs
```

Deploy to GitHub Pages:

```
1 $ foliant make ghp -p my-project✓
2 Parsing config✓
3 Applying preprocessor mkdocs✓
```

```
4 Making ghp with MkDocs—————
5
6 Result: https://account-name.github.io/my-project/
```

Config

You don't have to put anything in the config to use MkDocs backend. If it's installed, Foliant detects it.

To customize the output, use options in `backend_config.mkdocs` section:

```
1 backend_config:
2   mkdocs:
3     binary_path: mkdocs
4     use_title: true
5     use_chapters: true
6     mkdocs.yml:
7       site_name: Custom Title
8       site_url: http://example.com
9       site_author: John Smith
```

binary_path Path to the MkDocs executable. By default, `mkdocs` command is run, which implies it's somewhere in your `PATH`.

use_title If `true`, use `title` value from `foliant.yml` as `site_name` in `mkdocs.yml`. In this case, you don't have to specify `site_name` in `mkdocs.yml` section. If you do, the value from `mkdocs.yml` section has higher priority. If `false`, you *must* specify `site_name` manually, otherwise MkDocs will not be able to build the site.

Default is `true`.

use_chapters Similar to `use_title`, but for pages. If `true`, `chapters` value from `foliant.yml` is used as `pages` in `mkdocs.yml`.

mkdocs.yml Params to be copied into `mkdocs.yml` file. The params are passed "as is," so you should consult with the [MkDocs configuration docs](#).

Preprocessor

MkDocs backend ships with a preprocessor that transforms a Foliant project into a MkDocs one. Basically, `foliant make mkdocs` just applies the preprocessor.

The preprocessor is invoked automatically when you run MkDocs backend, so you don't have to add it in `preprocessors` section manually.

However, it's just a regular preprocessor like any other, so you can call it manually if necessary:

```
1 preprocessors:  
2   - mkdocs:  
3     mkdocs_project_dir_name: mkdocs
```

mkdocs_project_dir_name Name of the directory for the generated MkDocs project within the tmp directory.

Troubleshooting

Fenced Code Is Not Rendered in List Items or Blockquotes

MkDocs can't handle fenced code blocks in blockquotes or list items due to an [issue in Python Markdown](#).

Unfortunately, nothing can be done about it, either on MkDocs's or Foliant's part. As a workaround, use [indented code blocks](#).

Paragraphs Inside List Items Are Rendered on the Root Level

Check if you use **four-space indentation**. [Python Markdown is stern about this point](#).

Pandoc

[Pandoc](#) is a Swiss-army knife document converter. It converts almost any format to any other format: md to pdf, rst to html, adoc to docx, and so on and so on.

Pandoc backend for Foliant add `pdf`, `docx`, and `tex` targets.

Installation

```
$ pip install foliantcontrib.pandoc
```

You also need to install Pandoc and TeXLive distribution for your platform.

Usage

Build pdf:

```
1 $ foliant make pdf -p my-project✓  
2 Parsing config✓  
3 Applying preprocessor flatten✓  
4 Making pdf with Pandoc—————  
5  
6 Result: My_Project-2017-12-04.pdf
```

Build docx:

```
1 $ foliant make docx -p my-project✓  
2 Parsing config✓  
3 Applying preprocessor flatten✓  
4 Making docx with Pandoc—————  
5  
6 Result: My_Project-2017-12-04.docx
```

Build tex (mostly for pdf debugging):

```
1 $ foliant make tex -p my-project✓  
2 Parsing config✓  
3 Applying preprocessor flatten✓  
4 Making docx with Pandoc—————  
5  
6 Result: My_Project-2017-12-04.tex
```

Config

You don't have to put anything in the config to use Pandoc backend. If it's installed, Foliant will detect it.

You can however customize the backend with options in `backend_config.pandoc` section:

```
1 backend_config:  
2   pandoc:
```

```

3   pandoc_path: pandoc
4   template: !path template.tex
5   vars:
6       ...
7   reference_docx: !path reference.docx
8   params:
9       ...
10  filters:
11      ...
12  markdown_flavor: markdown
13  markdown_extensions:
14      ...

```

pandoc_path is the path to `pandoc` executable. By default, it's assumed to be in the `PATH`.

template is the path to the TeX template to use when building pdf and tex (see [“Templates”](#) in the Pandoc documentation).

Tip

Use `!path` tag to ensure the value is converted into a valid path relative to the project directory.

vars is a mapping of template variables and their values.

reference_docx is the path to the reference document to used when building docx (see [“Templates”](#) in the Pandoc documentation).

Tip

Use `!path` tag to ensure the value is converted into a valid path relative to the project directory.

params are passed to the `pandoc` command. Params should be defined by their long names, with dashes replaced with underscores (e.g. `--pdf-engine` is defined as `pdf_engine`).

filters is a list of Pandoc filters to be applied during build.

markdown_flavor is the Markdown flavor assumed in the source. Default is `markdown`, [Pandoc's extended Markdown](#). See [“Markdown Variants”](#) in the Pandoc documentation.

markdown_extensions is a list of Markdown extensions applied to the Markdown source. See [Pandoc's Markdown](#) in the Pandoc documentation.

Example config:

```
1 backend_config:
2   pandoc:
3     template: !path templates/basic.tex
4     vars:
5       toc: true
6       title: This Is a Title
7       second_title: This Is a Subtitle
8       logo: !path templates/logo.png
9       year: 2017
10    params:
11      pdf_engine: xelatex
12      listings: true
13      number_sections: true
14    markdown_extensions:
15      - simple_tables
16      - fenced_code_blocks
17      - strikeout
```

Troubleshooting

Could not convert image ...: check that rsvg2pdf is in path

In order to use svg images in pdf, you need to have `rsvg-convert` executable in `PATH`.

On macOS, `brew install librsvg` does the trick. On Ubuntu, `apt install librsvg2-bin`. On Windows, [download rsvg-convert.7z](#) (without fontconfig support), unpack `rsvg-convert.exe`, and put it anywhere in `PATH`. For example, you can put it in the same directory where you run `foliant make`.

Preprocessors

Blockdiag

Blockdiag is a tool to generate diagrams from plain text. This preprocessor finds diagram definitions in the source and converts them into images on the fly during project build. It supports all Blockdiag flavors: blockdiag, seqdiag, actdiag, and nwdiag.

Installation

```
$ pip install foliantcontrib.blockdiag
```

Config

To enable the preprocessor, add `blockdiag` to `preprocessors` section in the project config:

```
1 preprocessors:  
2   - blockdiag
```

The preprocessor has a number of options:

```
1 preprocessors:  
2   - blockdiag:  
3       cache_dir: !path .diagramscache  
4       blockdiag_path: blockdiag  
5       seqdiag_path: seqdiag  
6       actdiag_path: actdiag  
7       nwdiag_path: nwdiag  
8       params:  
9         ...
```

cache_dir Path to the directory with the generated diagrams. It can be a path relative to the project root or a global one; you can use `~/` shortcut.

Note

To save time during build, only new and modified diagrams are rendered. The generated images are cached and reused in future builds.

***_path** Paths to the `blockdiag`, `seqdiag`, `actdiag`, and `nwdiag` binaries. By default, it is assumed that you have these commands in `PATH`, but if they're installed in a custom place, you can define it here.

params Params passed to the image generation commands (`blockdiag`, `seqdiag`, etc.). Params should be defined by their long names, with dashes replaced with underscores (e.g. `--no-transparency` becomes `no_transparency`); also, `-T` param is called `format` for readability:

```
1 preprocessors:
2   - blockdiag:
3     params:
4       antialias: true
5       font: !path Anonymous_pro.ttf
```

To see the full list of params, run `blockdiag -h`.

Usage

To insert a diagram definition in your Markdown source, enclose it between `<<blockdiag>...</blockdiag>`, `<<seqdiag>...</seqdiag>`, `<actdiag>...</actdiag>`, or `<nwdiag>...</nwdiag>` tags (indentation inside tags is optional):

```
1 Here's a block diagram:
2
3 <<blockdiag>
4   blockdiag {
5     A -> B -> C -> D;
6     A -> E -> F -> G;
7   }
8 </blockdiag>
9
10 Here's a sequence diagram:
11
12 <<seqdiag>
13   seqdiag {
14     browser -> webserver [label = "GET /index.html"];
15     browser <-- webserver;
16     browser -> webserver [label = "POST /blog/comment"];
```

```

17         webserver -> database [label = "INSERT
comment"];
18         webserver <-- database;
19     browser <-- webserver;
20 }
21 </seqdiag>

```

To set a caption, use `caption` option:

```

1 Diagram with a caption:
2
3 <<blockdiag caption="Sample diagram from the official site">
4     blockdiag {
5         A -> B -> C -> D;
6         A -> E -> F -> G;
7     }
8 </blockdiag>

```

You can override `params` values from the preprocessor config for each diagram:

```

1 By default, diagrams are in png. But this diagram is in svg:
2
3 <<blockdiag caption="High-quality diagram" format="svg">
4     blockdiag {
5         A -> B -> C -> D;
6         A -> E -> F -> G;
7     }
8 </blockdiag>

```

Flags

This preprocessors lets you exclude parts of the source based on flags defined in the project config and environment variables, as well as current target and backend.

Installation

```
$ pip install foliantcontrib.flags
```

Config

Enable the preprocessor by adding it to `preprocessors`:

```
1 preprocessors:
2   - flags
```

Enabled project flags are listed in `preprocessors.flags`:

```
1 preprocessors:
2   - flags:
3     - foo
4     - bar
```

To set flags for the current session, define `FOLIANT_FLAGS` environment variable:

```
$ FOLIANT_FLAGS="spam, eggs"
```

You can use commas, semicolons, or spaces to separate flags.

Hint

To emulate a particular target or backend with a flag, use the special flags `target:FLAG` and `backend:FLAG` where `FLAG` is your target or backend: `$ FOLIANT_FLAGS="target:pdf, backend:pandoc, spam"`

Usage

Conditional blocks are enclosed between `<if>...</if>` tags:

```
1 This paragraph is for everyone.
2
3 <if flags="management">
4 This paragraph is for management only.
5 </if>
```

A block can depend on multiple flags. You can pick whether all tags must be present for the block to appear, or any of them (by default, `kind="all"` is assumed):

```
1 <if flags="spam, eggs" kind="all">
2 This is included only if both `spam` and `eggs` are set.
3 </if>
4
```



```
5 <if flags="spam, eggs" kind="any">
6 This is included if both `spam` or `eggs` is set.
7 </if>
```

You can also list flags that must *not* be set for the block to be included:

```
1 <if flags="spam, eggs" kind="none">
2 This is included only if neither `spam` nor `eggs` are set.
3 </if>
```

You can check against the current target and backend instead of manually defined flags:

```
1 <if targets="pdf">This is for pdf output</if><if targets="
  site">This is for the site</if>
2
3 <if backends="mkdocs">This is only for MkDocs.</if>
```

Flatten

This preprocessor converts a Foliant project source directory into a single Markdown file containing all the sources, preserving order and inheritance.

This preprocessor is used by backends that require a single Markdown file as input instead of a directory. The Pandoc backend is one such example.

Installation

```
$ pip install foliantcontrib.flatten
```

Config

This preprocessor is required by Pandoc backend, so if you use it, you don't need to install Flatten or enable it in the project config manually.

However, it's still a regular preprocessor, and you can run it manually by listing it in preprocessors:

```
1 preprocessors:
2   - flatten
```

The preprocessor has only one option— `flat_src_file_name`. It's the name of the flattened file that is created in the tmp directory:

```
1 preprocessors:
2   - flatten:
3     flat_src_file_name: flattened.md
```

Default value is `__all__.md`.

Note

Flatten preprocessor uses includes, so when you install Pandoc backend, Includes preprocessor will also be installed, along with Flatten.

Includes

Includes preprocessor lets you reuse parts of other documents in your Foliant project sources. It can include from files on your local machine and remote Git repositories. You can include entire documents as well as parts between particular headings, removing or normalizing included headings on the way.

Installation

```
$ pip install foliantcontrib.includes
```

Config

To enable the preprocessor with default options, add `includes` to `preprocessors` section in the project config:

```
1 preprocessors:
2   - includes
```

The preprocessor has a number of options:

```
1 preprocessors:
2   - includes:
3     cache_dir: !path .includescache
4     recursive: true
5     aliases:
6     ...
```

cache_dir Path to the directory for cloned repositories. It can be a path relative to the project path or a global one; you can use `~/` shortcut.

Note

To include files from remote repositories, the preprocessor clones them. To save time during build, cloned repositories are stored and reused in future builds.

recursive Flag that defines whether includes in included documents should be processed.

aliases Mapping from aliases to Git repository URLs. Once defined here, an alias can be used to refer to the repository instead of its full URL.

For example, if you set this alias in the config:

```
1 - includes:
2   aliases:
3     foo: https://github.com/boo/bar.git
```

you can include README.md file content from this repository using this syntax:

```
<include>$foo$path/to/doc.md</include>
```

Usage

To include a document from your machine, put the path to it between `<include>...</include>` tags:

```
1 Text below is taken from another document.
2
3 <include>/path/to/another/document.md</include>
```

To include a document from a remote Git repository, put its URL between `$s` in front of the document path:

```
1 Text below is taken from a remote repository.
2
3 <include>
4   $https://github.com/foo/bar.git$path/to/doc.md
5 </include>
```

If the repository alias is defined in the project config, you can use it instead of the URL:

```
1 - includes:
2   aliases:
3     foo: https://github.com/foo/bar.git
```

And then in the source:

```
<include>$foo$path/to/doc.md</include>
```

You can also specify a particular branch or revision:

```
1 Text below is taken from a remote repository on branch
  develop.
2
3 <include>$foo#develop$path/to/doc.md</include>
```

To include a part of a document between two headings, use the `#Start:Finish` syntax after the file path:

```
1 Include content from "Intro" up to "Credits":
2
3 <include>sample.md#Intro:Credits</include>
4
5 Include content from start up to "Credits":
6
7 <include>sample.md#:Credits</include>
8
9 Include content from "Intro" up to the next heading of the
  same level:
10
11 <include>sample.md#Intro</include>
```

Options

sethead The level of the topmost heading in the included content. Use it to guarantee that the included text doesn't break the parent document's heading order:

```
1 # Title
```

```

2
3 ## Subtitle
4
5 <include sethead="3">
6     other.md
7 </include>

```

nohead Flag that tells the preprocessor to strip the starting heading from the included content:

```

1 # My Custom Heading
2
3 <include nohead="true">
4     other.md#Original Heading
5 </include>

```

Default is `false`.

Options can be combined. For example, use both `sethead` and `nohead` if you want to include a section with a custom heading:

```

1 ## My Custom Heading
2
3 <include sethead="1" nohead="true">
4     other.md#Original Heading
5 </include>

```

Macros

Macro is a string with placeholders that is replaced with predefined content during documentation build. Macros are defined in the config.

Installation

```
$ pip install foliantcontrib.macros
```

Config

Enable the preprocessor by adding it to `preprocessors` and listing your macros in `macros` dictionary:

```

1 preprocessors:
2   - macros:
3     macros:
4       foo: This is a macro definition.
5       bar: "This is macro with a parameter: {0}"

```

Usage

Here's the simplest usecase for macros:

```

1 preprocessors:
2   - macros:
3     macros:
4       support_number: "8 800 123-45-67"

```

Now, every time you need to insert your support phone number, you put a macro instead:

```

1 Call you support team: <macro>support_number</macro>.
2
3 Here's the number again: <<m>support_number</m>.

```

Macros are useful in documentation that should be built into multiple targets, e.g. site and pdf, when the same thing is done differently in one target than in the other.

For example, to reference a page in MkDocs, you just put the Markdown file in the link:

```
Here is [another page](another_page.md).
```

But when building documents with Pandoc all sources are flattened into a single Markdown, so you refer to different parts of the document by anchor links:

```
Here is [another page](#another_page).
```

This can be implemented using `<if></if>` tag:

```
Here is [another page](#another_page).
```

This bulky construct quickly gets old when you use many cross-references in your documentation.

To make your sources cleaner, move this construct to the config as a reusable macro:

```
1 preprocessors:  
2   - macros:  
3     macros:  
4       ref: <if backends="pandoc">{0}</if>
```

And use it in the source:

```
Here is [another page](<macro params="#another_page,  
another_page.md">ref</macro>).
```

CLI Extensions

Init

This CLI extension add `init` command that lets you create Foliant projects from templates.

Installation

```
$ pip install foliantcontrib.init
```

Usage

Create project from the default “basic” template:

```
1 $ foliant init
2 Enter the project name: Awesome Docs✓
3 Generating Foliant project—————
4
5 Project "Awesome Docs" created in awesome-docs
```

Create project from a custom template:

```
1 $ foliant init --template /path/to/custom/template
2 Enter the project name: Awesome Customized Docs✓
3 Generating Foliant project—————
4
5 Project "Awesome Customized Docs" created in awesome-
  customized-docs
```

You can provide the project name without user prompt:

```
1 $ foliant init --name Awesome Docs✓
2 Generating Foliant project—————
3
4 Project "Awesome Docs" created in awesome-docs
```

Another useful option is `--quiet`, which hides all output except for the path to the generated project:

```
1 $ foliant init --name Awesome Docs --quiet
2 awesome-docs
```

To see all available options, run `foliant init --help`:

```
1 $ foliant init --help
2 usage: foliant init [-h] [-n NAME] [-t NAME or PATH] [-q]
3
4 Generate new Foliant project.
5
6 optional arguments:
7   -h, --help            show this help message and exit
8   -n NAME, --name NAME  Name of the Foliant project
9   -t NAME or PATH, --template NAME or PATH
10                        Name of a built-in project template
                        or path to custom one
11   -q, --quiet           Hide all output except for the
                        result. Useful for piping.
```

Project Templates

A project template is a regular Foliant project but containing placeholders in files. When the project is generated, the placeholders are replaced with the values you provide. Currently, there are two placeholders: `{title}` and `{slug}`.

There is a built-in template called `basic`. It's used by default if no template is specified.