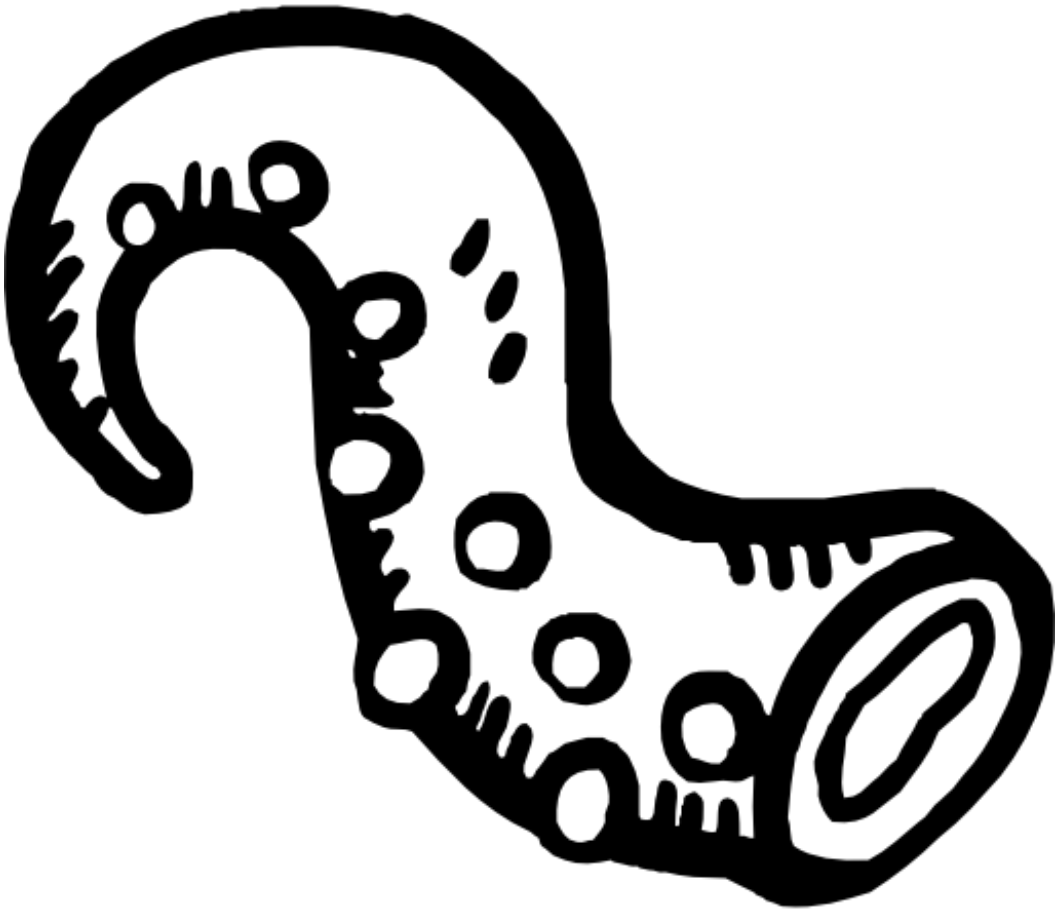


Foliant

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:

```
$ add-apt-repository ppa:jonathonf/python-3.6
$ 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:

```
$ apt update && apt install -y wget texlive-full librsvg2
-bin
$ 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:

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

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

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

Then, run Foliant in a container:

```
# Site:
$ docker run --rm -it -v `pwd`:/usr/src/app -w /usr/src/app my-project make site
# Pdf:
$ 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.

To use it, run `foliant init` command:

```
$ foliant init
Enter the project name: Hello Foliant
v Generating Foliant project──────────────────────

Project "Hello Foliant" created in hello-foliant
```

To do the same with Docker, run:

```
$ docker run --rm -it -v `pwd`:/usr/app/src -w /usr/app/src
foliant/foliant init
Enter the project name: Hello Foliant
v Generating Foliant project──────────────────────

Project "Hello Foliant" created in hello-foliant
```

Here's what this command created:

```
$ cd hello-foliant
$ tree
.├──
 docker-compose.yml├──
 Dockerfile├──
 foliant.yml├──
 README.md├──
 requirements.txt└──
```

```
src
   └── index.md

1 directory, 6 files
```

`foliant.yml` is your project's config file.

`src` directory is where the content of the project lives. Currently, there's just one file `index.md`.

`requirements.txt` lists the Python packages required for the project: Foliant backends and preprocessors, MkDocs themes, and what not. The the Docker image for the project is built, these requirements are installed in it.

`Dockerfile` and `docker-compose.yml` are necessary to build the project in a Docker container.

Build Site

In the project directory, run:

```
$ foliant make site
v Parsing config
v Applying preprocessor mkdocs
v Making site with MkDocs──────────────────────

Result: Hello_Foliant-2018-01-23.mkdocs
```

Or, with Docker Compose:

```
$ docker-compose run --rm hello-foliant make site
v Parsing config
v Applying preprocessor mkdocs
v Making site with MkDocs──────────────────────

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:

```
$ cd Hello_Foliant-2018-01-23.mkdocs
$ python -m http.server
```

```
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:

```
$ foliant make pdf
v Parsing config
v Applying preprocessor flatten
v Making pdf with Pandoc──────────────────────

Result: Hello_Foliant-2018-01-23.pdf
```

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

```
- FROM foliant/foliant
+ # FROM foliant/foliant
# If you plan to bake PDFs, uncomment this line and comment the line above:
- # FROM foliant/foliant:pandoc
+ FROM foliant/foliant:pandoc

COPY requirements.txt .

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. Also, run it whenever you need to update the versions of the required packages from `requirements.txt`.

Then, run this command in the project directory:

```
$ docker-compose run --rm hello-foliant make pdf
v Parsing config
v Applying preprocessor flatten
v Making pdf with Pandoc————————————————————

Result: Hello_Foliant-2018-01-23.pdf
```

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

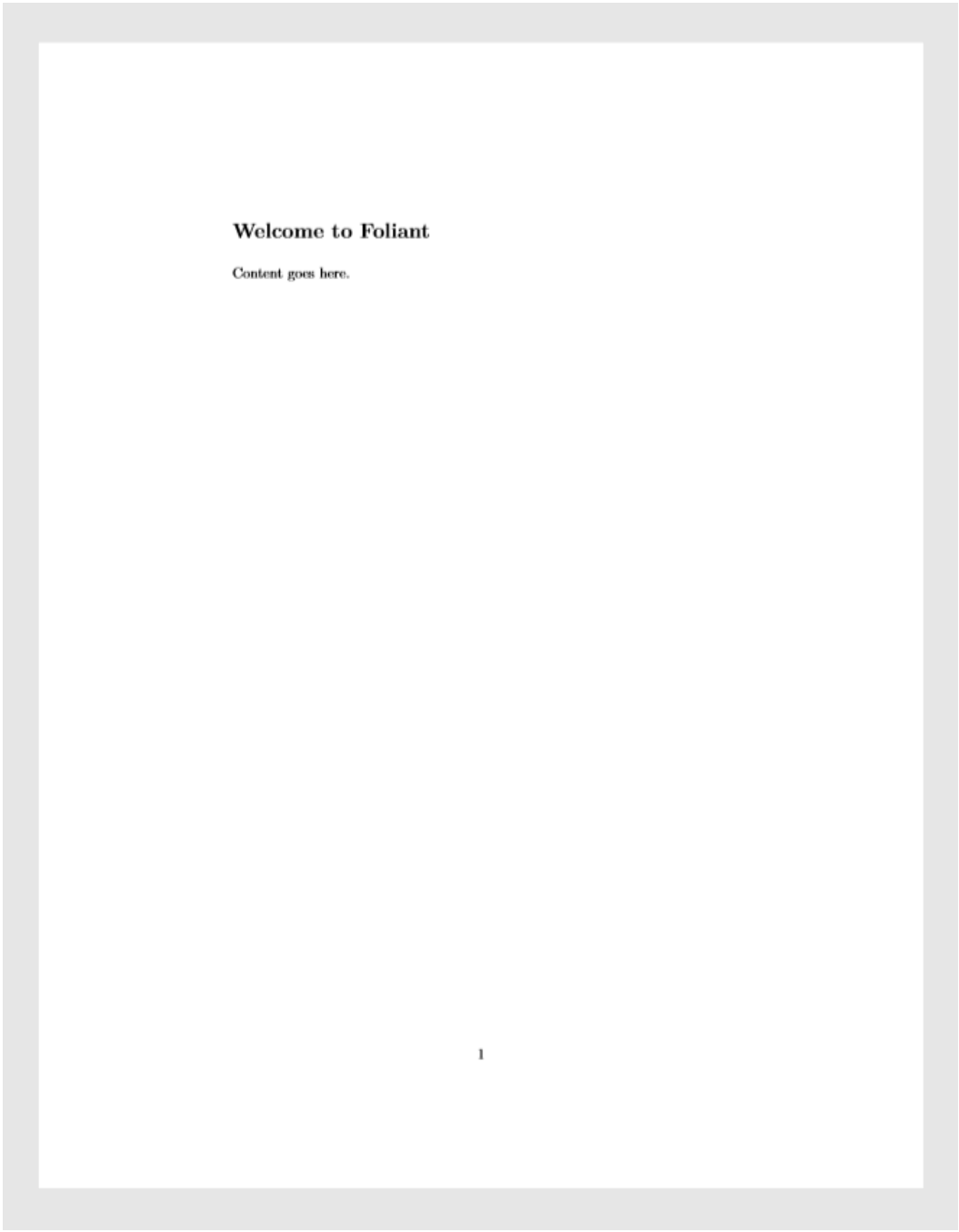


Figure 2. Basic Foliant project built with Pandoc

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:

```
# Hello Again

This is regular text generated from regular Markdown.

Foliant doesn't force any *special* Markdown flavor.
```

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

```
title: Hello Foliant

chapters:
  - index.md
+ - hello.md
```

Rebuild the project to see the new page:

```
$ docker-compose run --rm hello-foliant make site && docker-compose run --rm hello-foliant make pdf
v Parsing config
v Applying preprocessor mkdocs
v Making site with MkDocs————————————————————

Result: Hello_Foliant-2018-02-08.mkdocs
v Parsing config
v Applying preprocessor flatten
v Making pdf with Pandoc—————————————————————

Result: Hello_Foliant-2018-02-08.pdf
```

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

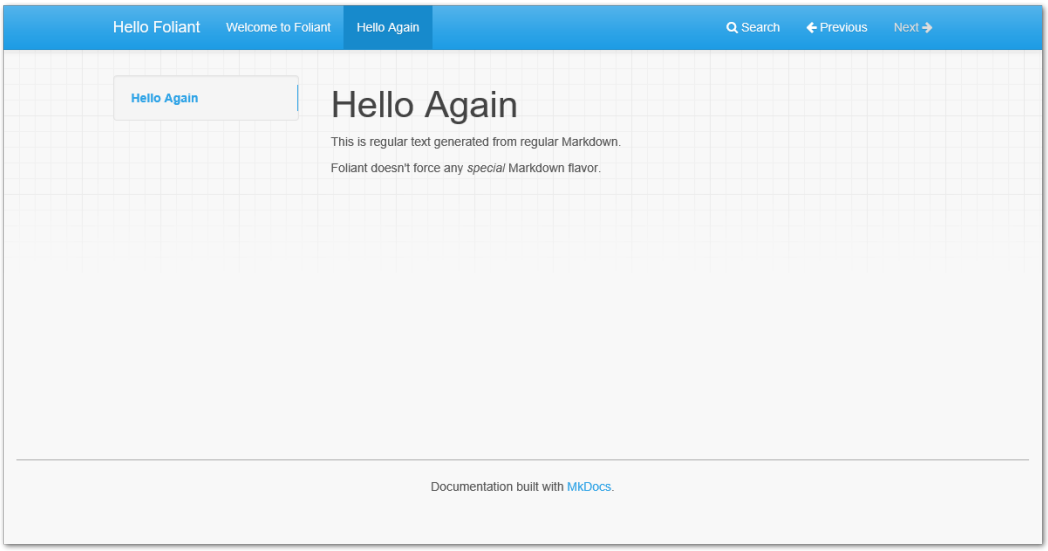


Figure 3. New page on the site

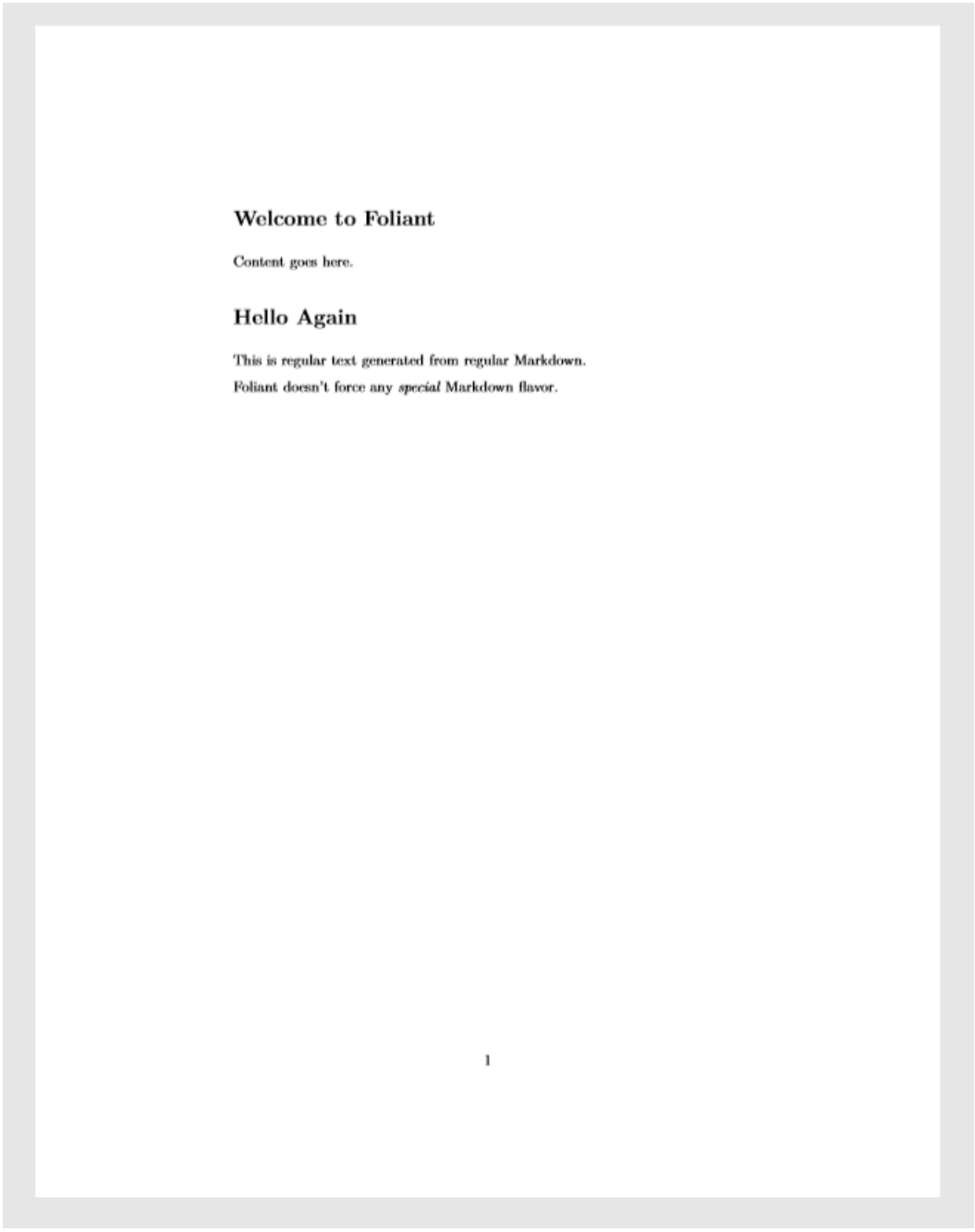


Figure 4. New page in the pdf document

Use Preprocessors

Preprocessors is what makes Foliant special and extremely useful. Preprocessors are additional packages that, well, preprocess the source code of your project. You can do all kinds of stuff with preprocessors:

— include remote Markdown files or their parts in the source files
— render diagrams from textual description on build
— restructure the project source or compile it into a single file for a particular backend

In fact, you have already used two preprocessors! Check the output of the `foliant make` commands and note the lines `v Applying preprocessor mkdocs` and `v Applying preprocessor flatten`. The first one informs you that the project source has been preprocessed with `mkdocs` preprocessor in order to make it compatible with MkDocs' requirements, and the second one tells you that `flatten` preprocessor was used to squash the project source into one a single file (because Pandoc only works with single files).

These preprocessors where called by MkDocs and Pandoc backends respectively. You didn't have to install or activate them explicitly.

Now, let's try to use a different kind of preprocessors, the ones that register new tags: Blockdiag.

Embed Diagrams with Blockdiag

Blockdiag is a Python app for generating diagrams. Blockdiag preprocessor extracts diagram descriptions from the project source and replaces them with the generated images.

In `hello.md`, add the following lines:

```
Foliant doesn't force any *special* Markdown flavor.

+ <seqdiag caption="This diagram is generated on the fly">
+   seqdiag {
+     "foliant make site" -> "mkdocs preprocessor" -> "
blockdiag preprocessor" -> "mkdocs backend" -> site;
+   }
+ </seqdiag>
```

Blockdiag preprocessor adds several tags to Foliant, each corresponding to a certain diagram type. Sequence diagrams are defined with `<<seqdiag></seqdiag>` tag. This is what we used in the sample above. The diagram definition sits in the tag body and the diagram properties such as caption or format are defined as tag parameters.

Rebuild the site with `foliant make site` and open it in the browser:

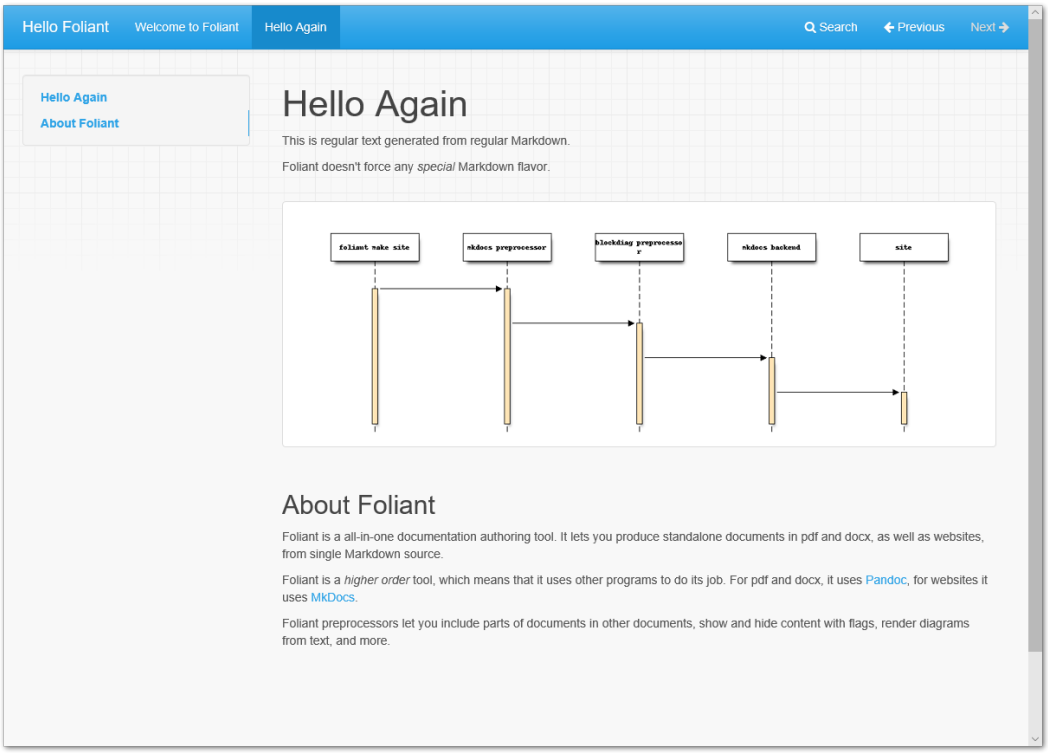
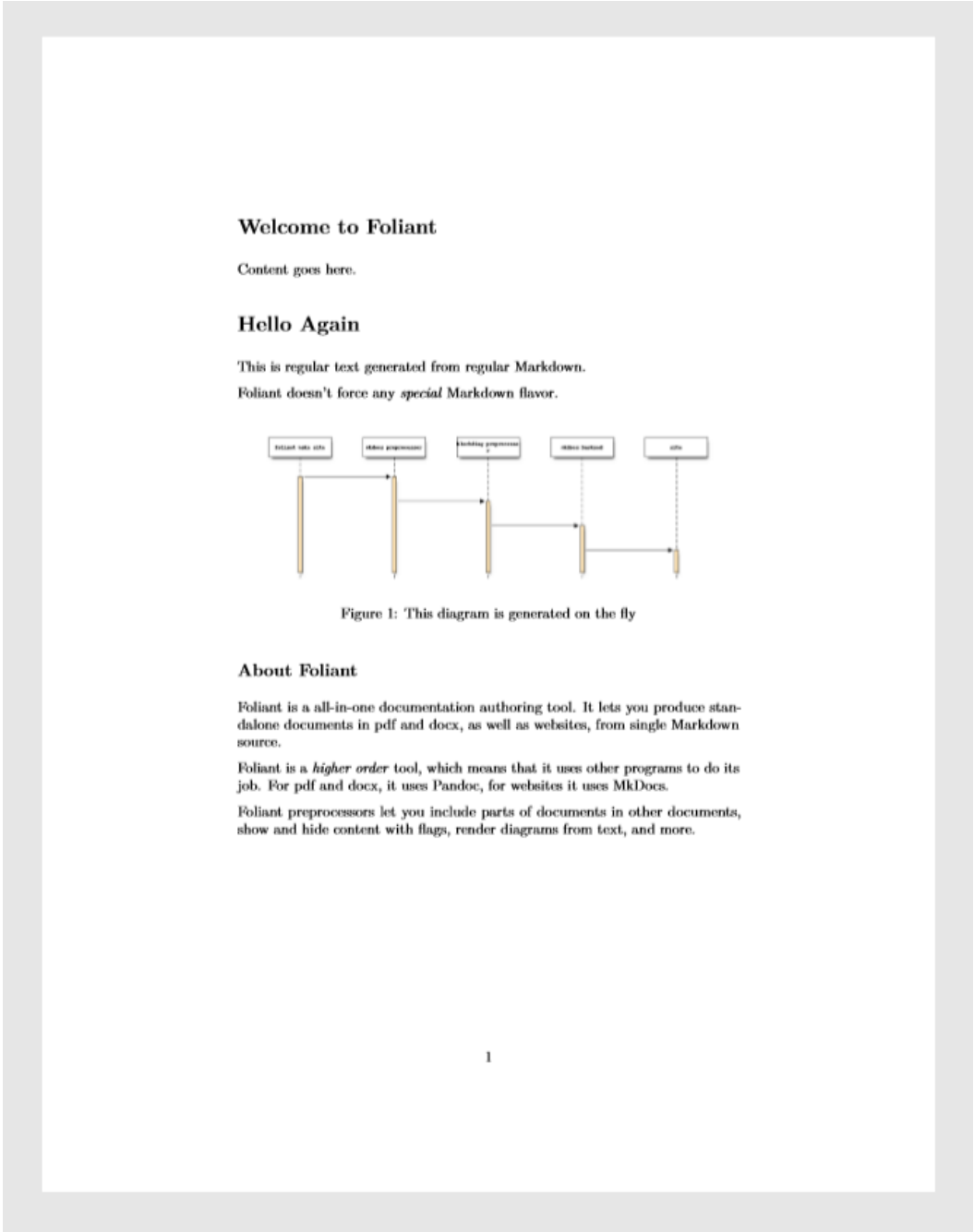


Figure 5. Sequence diagram drawn with seqdiag on the site

Rebuild the pdf as well and see that the diagram there:



Welcome to Foliant

Content goes here.

Hello Again

This is regular text generated from regular Markdown.

Foliant doesn't force any *special* Markdown flavor.

Figure 1: This diagram is generated on the fly

About 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 that 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 more.

1

Figure 6. Sequence diagram drawn with seqdiag in the pdf

Let's customize the look of the diagrams in our project by setting their properties in the config file. For example, let's use a custom font for labels. I'm using the ever popular Comic Sans font, but you can pick any font that's available in `.ttf` format.

Put the font file in the project directory and add the following lines to `foliant.yml`:

```
preprocessors:
-    - blockdiag
+    - blockdiag:
+        params:
+            font: !path comic.ttf
```

After a rebuild, the diagram on the site and in the pdf should look like this:

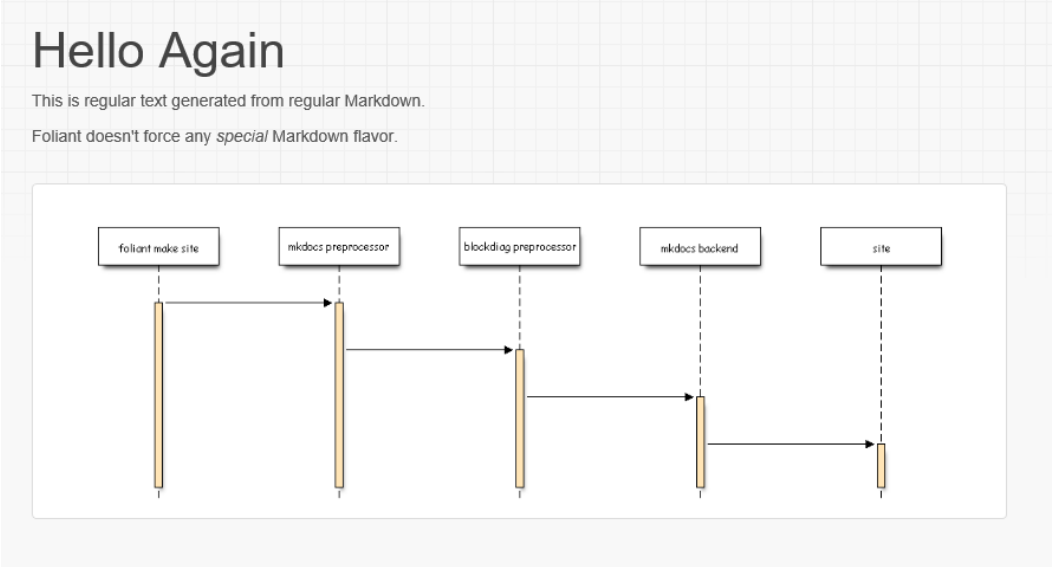


Figure 7. Sequence diagram with Comic Sans in labels, site

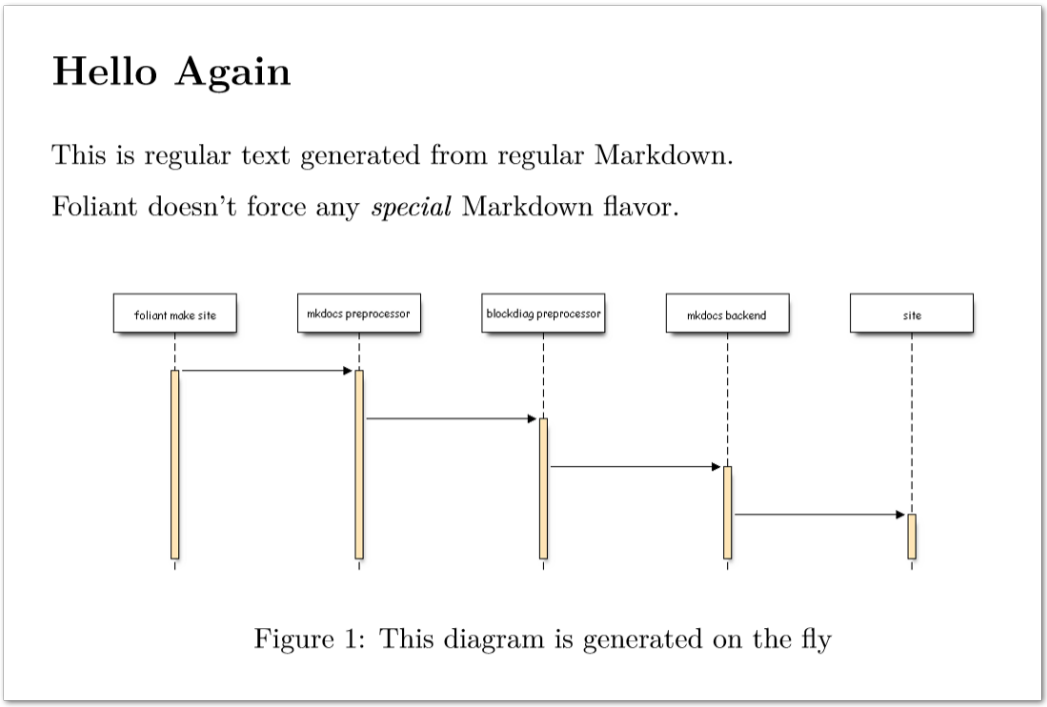


Figure 8. Sequence diagram with Comic Sans in labels, pdf

There are many more params you can define for your diagrams. You can override global params for particluar diagrams in their tags. And by combining this preprocessor with Flags you can even set different params for different backends, for example build vector diagrams for pdf output and bitmap for site:

```
This is a diagram that is rendered to `.png` in html and to `.pdf` in pdf:

<blockdiag format="pdf</if">
    ...
</blockdiag>
```

The possibilities acquired by combining different preprocessors are endless!

Why Foliant Uses XML syntax for Preprocessor Tags

It's common for Markdown-based tools to extend Markdown with custom syntax for additional functions. There's no standard for custom syntax in

the Markdown spec, so every developer uses whatever syntax is available for them, different one for every new extension.

In Foliant, we tried our best not to dive into this mess. Foliant aims to be an extensible platform, with many available preprocessors. So we needed one syntax for all preprocessors, but the one that was flexible enough to support them all.

After trying many options, we settled with XML. Yes, normally you'd have a nervous tick when you hear XML, and so would we, but this is one rare case where XML syntax belongs just right:

— it allows to provide tag body and named parameters
— it's familiar to every techwriter out there
— it's close enough to HTML, and HTML tags are actually allowed by the Markdown spec, so we're not even breaking the vanilla Markdown spec (almost)
— it's nicely highlighted in IDEs and text editors

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:

```
$ foliant make mkdocs -p my-project✔
 Parsing config✔
 Applying preprocessor mkdocs✔
 Making mkdocs with MkDocs──────────────────────

Result: My_Project-2017-12-04.mkdocs.src
```

Build a standalone website:

```
$ foliant make site -p my-project✔
 Parsing config✔
 Applying preprocessor mkdocs✔
 Making site with MkDocs────────────────────────

Result: My_Project-2017-12-04.mkdocs
```

Deploy to GitHub Pages:

```
$ foliant make ghp -p my-project✔
 Parsing config✔
 Applying preprocessor mkdocs✔
```

```
 Making ghp with MkDocs──────────────────────

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:

```
backend_config:
  mkdocs:
    binary_path: mkdocs
    use_title: true
    use_chapters: true
    mkdocs.yml:
      site_name: Custom Title
      site_url: http://example.com
      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`. It 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:

```
preprocessors:
  - mkdocs:
      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:

```
$ foliant make pdf -p my-project✔
 Parsing config✔
 Applying preprocessor flatten✔
 Making pdf with Pandoc──────────────────────

Result: My_Project-2017-12-04.pdf
```

Build docx:

```
$ foliant make docx -p my-project✔
 Parsing config✔
 Applying preprocessor flatten✔
 Making docx with Pandoc─────────────────────

Result: My_Project-2017-12-04.docx
```

Build tex (mostly for pdf debugging):

```
$ foliant make tex -p my-project✔
 Parsing config✔
 Applying preprocessor flatten✔
 Making docx with Pandoc─────────────────────

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:

```
backend_config:
  pandoc:
```

```
  pandoc_path: pandoc
  template: !path template.tex
  vars:
    ...
  reference_docx: !path reference.docx
  params:
    ...
  filters:
    ...
  markdown_flavor: markdown
  markdown_extensions:
    ...
```

`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:

```
backend_config:
  pandoc:
    template: !path templates/basic.tex
    vars:
      toc: true
      title: This Is a Title
      second_title: This Is a Subtitle
      logo: !path templates/logo.png
      year: 2017
    params:
      pdf_engine: xelatex
      listings: true
      number_sections: true
    markdown_extensions:
      - simple_tables
      - fenced_code_blocks
      - 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:

```
preprocessors:
  - blockdiag
```

The preprocessor has a number of options:

```
preprocessors:
  - blockdiag:
      cache_dir: !path .diagramscache
      blockdiag_path: blockdiag
      seqdiag_path: seqdiag
      actdiag_path: actdiag
      nwdiag_path: nwdiag
      params:
        ...
```

`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:

```
preprocessors:
  - blockdiag:
      params:
        antialias: true
        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):

```
Here's a block diagram:

<<blockdiag>
  blockdiag {
    A -> B -> C -> D;
    A -> E -> F -> G;
  }
</blockdiag>

Here's a sequence diagram:

<<seqdiag>
  seqdiag {
    browser  -> webserver [label = "GET /index.html"];
    browser <-- webserver;
    browser  -> webserver [label = "POST /blog/comment"];
```

```
                webserver  -> database [label = "INSERT
comment"];
                webserver <-- database;
    browser <-- webserver;
  }
</seqdiag>
```

To set a caption, use `caption` option:

```
Diagram with a caption:

<<blockdiag caption="Sample diagram from the official site">
  blockdiag {
    A -> B -> C -> D;
    A -> E -> F -> G;
  }
</blockdiag>
```

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

```
By default, diagrams are in png. But this diagram is in svg:

<<blockdiag caption="High-quality diagram" format="svg">
  blockdiag {
    A -> B -> C -> D;
    A -> E -> F -> G;
  }
</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 propressor by adding it to `preprocessors`:

```
preprocessors:
  - flags
```

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

```
preprocessors:
  - flags:
    - foo
    - 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>...` tags:

```
This paragraph is for everyone.

<<if flags="management">
This parapraph is for management only.
</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):

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

```

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

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

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

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

```
<if targets="pdf">This is for pdf output</if><if targets="
site">This is for the site</if>

<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`:

```
preprocessors:
  - 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:

```
preprocessors:
  - flatten:
      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:

```
preprocessors:
  - includes
```

The preprocessor has a number of options:

```
preprocessors:
  - includes:
      cache_dir: !path .includescache
      recursive: true
      aliases:
        ...
```

`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:

```
- includes:
  aliases:
    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:

```
Text below is taken from another document.

<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:

```
Text below is taken from a remote repository.

<include>
    $https://github.com/foo/bar.git$path/to/doc.md
</include>
```

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

```
- includes:
    aliases:
      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:

```
Text below is taken from a remote repository on branch
develop.

<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:

```
Include content from "Intro" up to "Credits":

<include>sample.md#Intro:Credits</include>

Include content from start up to "Credits":

<include>sample.md#:Credits</include>

Include content from "Intro" up to the next heading of the
same level:

<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:

```
# Title
```

```

## Subtitle

<include sethead="3">
    other.md
</include>
```

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

```
# My Custom Heading

<include nohead="true">
    other.md#Original Heading
</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:

```
## My Custom Heading

<include sethead="1" nohead="true">
  other.md#Original Heading
</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:

```
preprocessors:
  - macros:
      macros:
        foo: This is a macro definition.
        bar: "This is macro with a parameter: {0}"
```

Usage

Here's the simplest usecase for macros:

```
preprocessors:
  - macros:
      macros:
        support_number: "8 800 123-45-67"
```

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

```
Call you support team: <macro>support_number</macro>.

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:

```
preprocessors:
  - macros:
      macros:
        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:

```
$ foliant init
Enter the project name: Awesome Docs✔
 Generating Foliant project──────────────────────

Project "Awesome Docs" created in awesome-docs
```

Create project from a custom template:

```
$ foliant init --template /path/to/custom/template
Enter the project name: Awesome Customized Docs✔
 Generating Foliant project──────────────────────

Project "Awesome Customized Docs" created in awesome-
customized-docs
```

You can provide the project name without user prompt:

```
$ foliant init --name Awesome Docs✔
 Generating Foliant project──────────────────────

Project "Awesome Docs" created in awesome-docs
```

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

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

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

```
$ foliant init --help
usage: foliant init [-h] [-n NAME] [-t NAME or PATH] [-q]

Generate new Foliant project.

optional arguments:
  -h, --help            show this help message and exit
  -n NAME, --name NAME  Name of the Foliant project
  -t NAME or PATH, --template NAME or PATH
                        Name of a built-in project template
or path to custom one
  -q, --quiet           Hide all output accept 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.