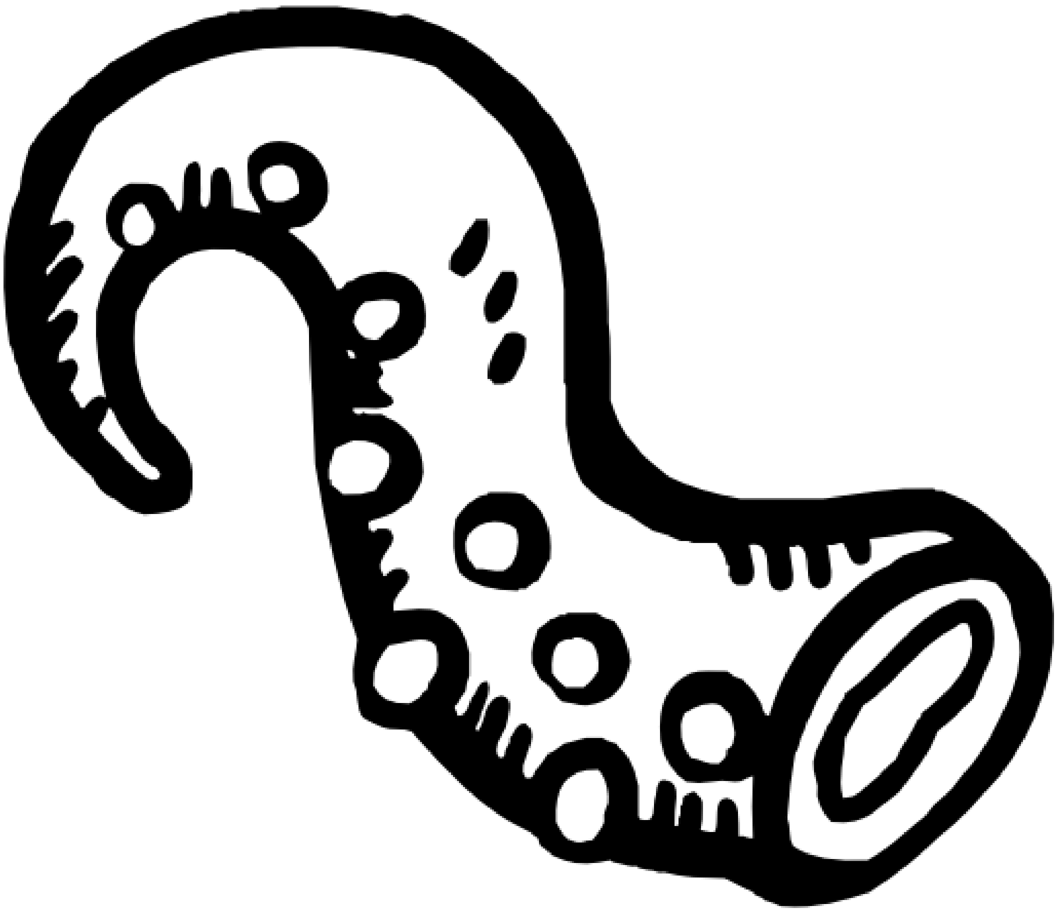


Foliant

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.

Who Is It for?

You'll love Foliant if you:

- need to ship documentation as pdf, docx, and website forms
- want to use Markdown with consistent extension system instead of custome syntax for every new bit of functionality
- like reStructuredText's extensibility and Asciidoc's flexibility, but actually would rather use Markdown
- want a tool that you can actually write custom extensions for without dealing with something as overengineered as Sphinx
- have documentation spread across many repos and want to reuse parts between documents

Changelog

1.0.10

- Add `escape_code` config option. To use it, escapecode and unescapecode preprocessors must be installed.

1.0.9

- Process attribute values of pseudo-XML tags as YAML.
- Allow single quotes for enclosing attribute values of pseudo-XML tags.
- Add `!project_path` and `!rel_path` YAML tags.

1.0.8

- Restore quiet mode.
- Add the `output()` method for using in preprocessors.

1.0.7

- Remove spinner made with Halo.
- Abolish quiet mode because it is useless if extensions are allowed to write anything to STDOUT.
- Show full tracebacks in debug mode; write full tracebacks into logs.

1.0.6

- CLI: If no args are provided, print help.
- Fix tags searching pattern in _unescape preprocessor.

1.0.5

- Allow to override default config file name in CLI.
- Allow multiline tags. Process `true` and `false` attribute values as boolean, not as integer.
- Add tests.
- Improve code style.

1.0.4

- **Breaking change.** Add logging to all stages of building a project. Config parser extensions, CLI extensions, backends, and preprocessors can now access `self.logger` and create child loggers with `self.logger = self.logger .getChild('newbackend')`.
- Add `pre` backend with `pre` target that applies the preprocessors from the config and returns a Foliant project that doesn't require any preprocessing.
- `make` now returns its result, which makes is easier to call it from extensions.

1.0.3

- Fix critical issue when config parsing would fail if any config value contained non-latin characters.

1.0.2

- Use README.md as package description.

1.0.1

- Fix critical bug with CLI module caused by missing version definition in the root `__init__.py` file.

1.0.0

- Complete rewrite.

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 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 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 with apt. On 14.04 and 16.04:

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

   On 14.04 and 16.04:

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

```
$ docker pull foliant/foliant
```

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✔
 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✔
 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✔
 Parsing config✔
 Applying preprocessor mkdocs✔
 Making site with MkDocs─────────────────────

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

Or, with Docker Compose:

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

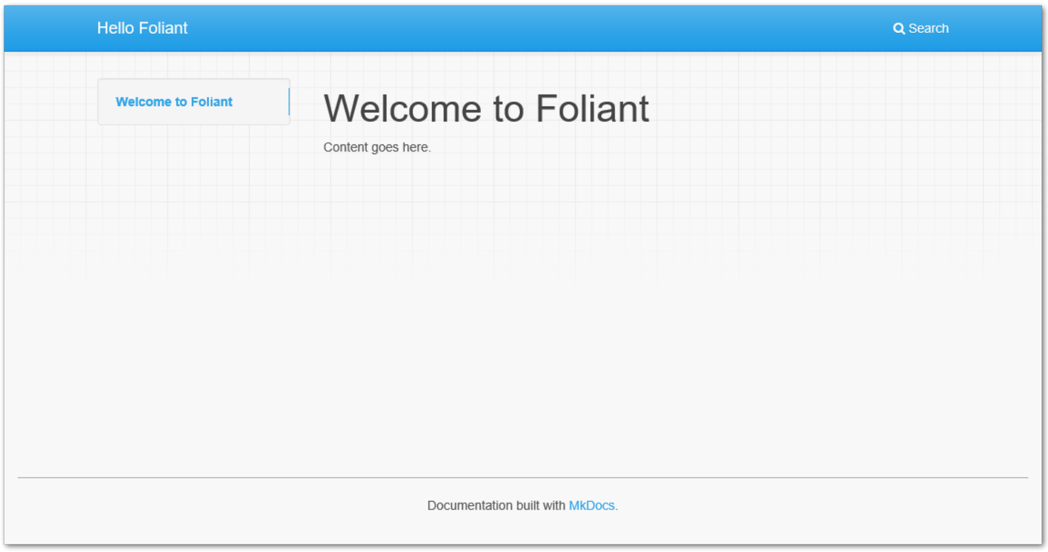
```

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:



Welcome to Foliant

Content goes here.

Documentation built with MkDocs.

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✔
 Parsing config✔
 Applying preprocessor flatten✔
 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
```

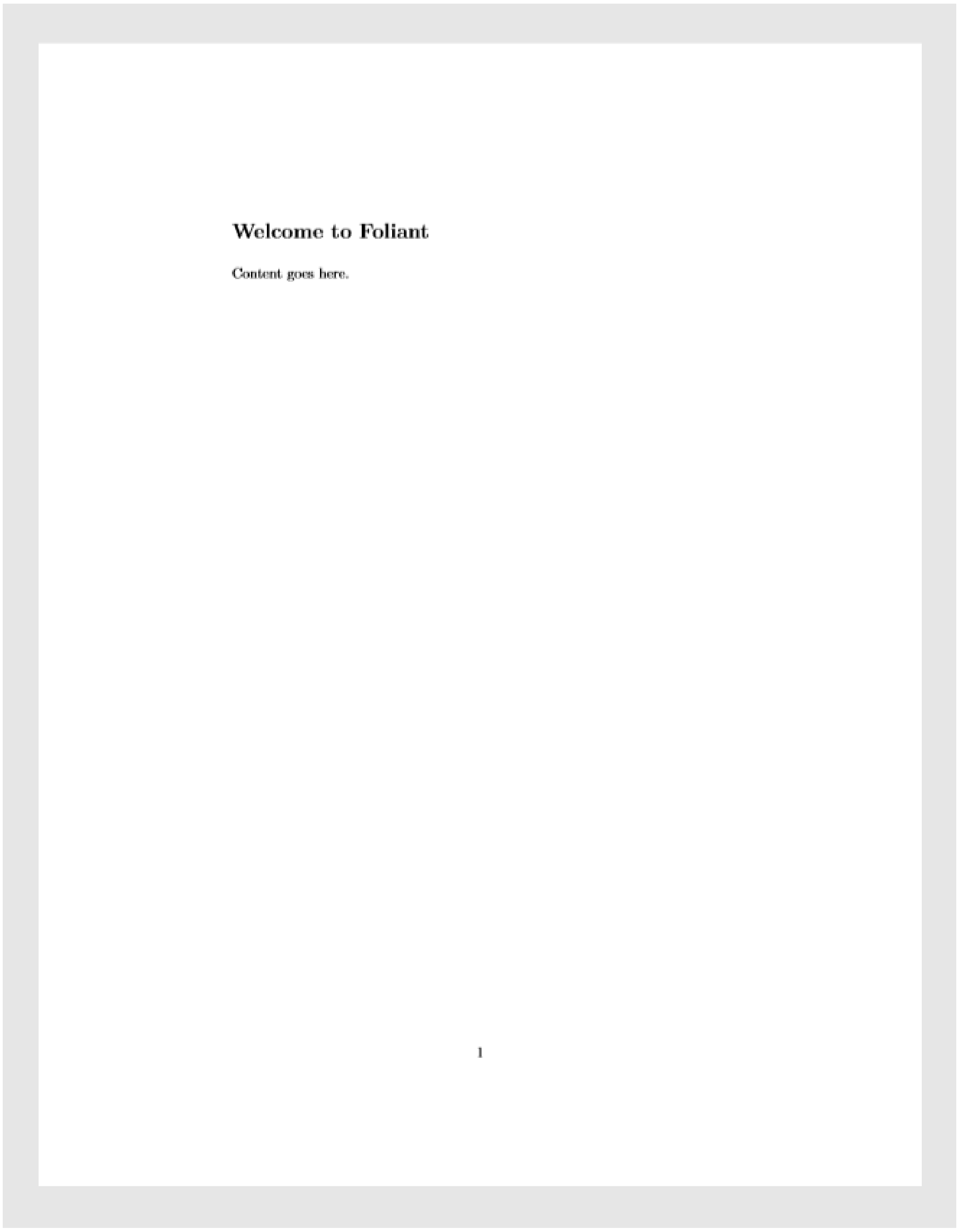
> **Note**
>
> Run `docker-compose build` to rebuild the image from the new base image if you have previously run `docker-compose run` with the old one. Also, run it whenever you need to update the versions of the required packages from `requirements.txt`.

Then, run this command in the project directory:

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

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

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


Welcome to Foliant

Content goes here.

1

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 'doesnt 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:

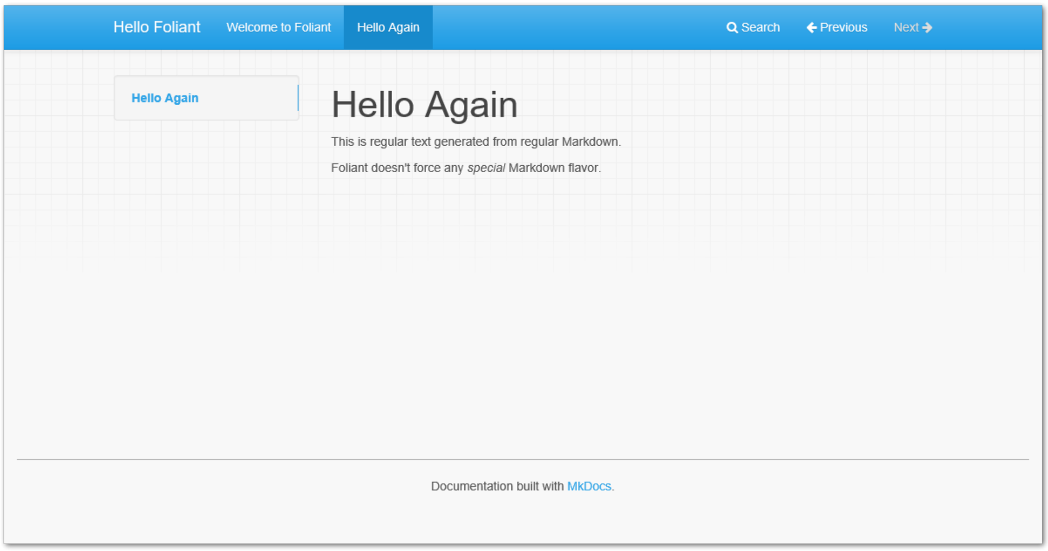
```
$ docker-compose run --rm hello-foliant make site && docker-compose run --rm hello-foliant make pdf✔
 Parsing config✔
 Applying preprocessor mkdocs✔
 Making site with MkDocs─────────────────────

Result: Hello_Foliant-2018-02-08.mkdocs✔
 Parsing config✔
 Applying preprocessor flatten✔
 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:



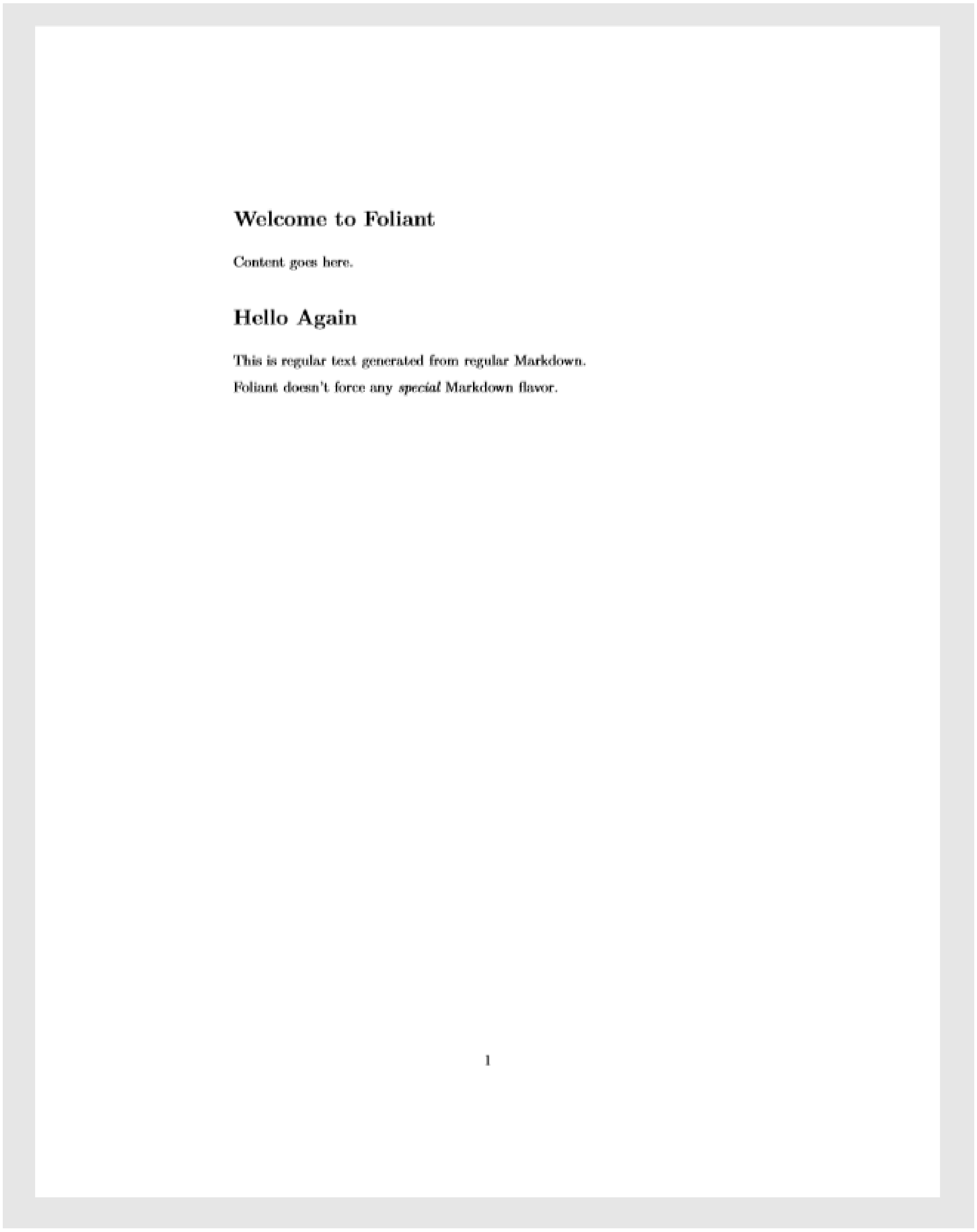
Hello Again

This is regular text generated from regular Markdown.

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

Documentation built with MkDocs.

Figure 3. New page on the site



Welcome to Foliant

Content goes here.

Hello Again

This is regular text generated from regular Markdown.

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

1

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 `Applying preprocessor mkdocs` and `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.

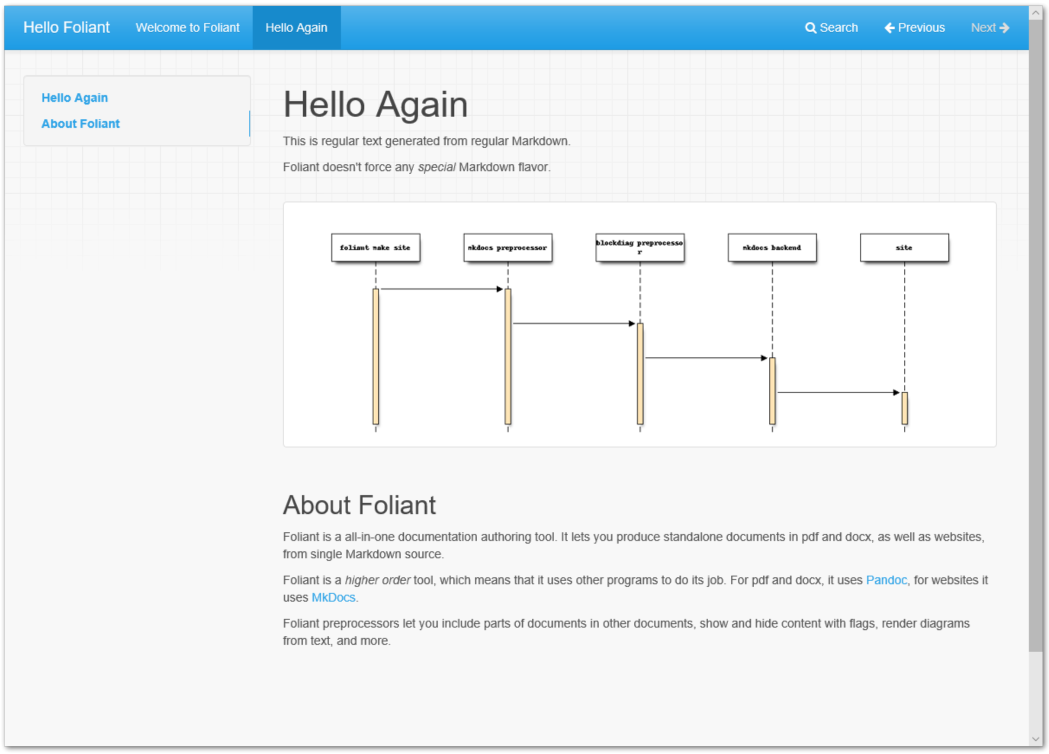
In `hello.md`, add the following lines:

```
Foliant 'doesnt 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:



Hello Again

This is regular text generated from regular Markdown.

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

About Foliant

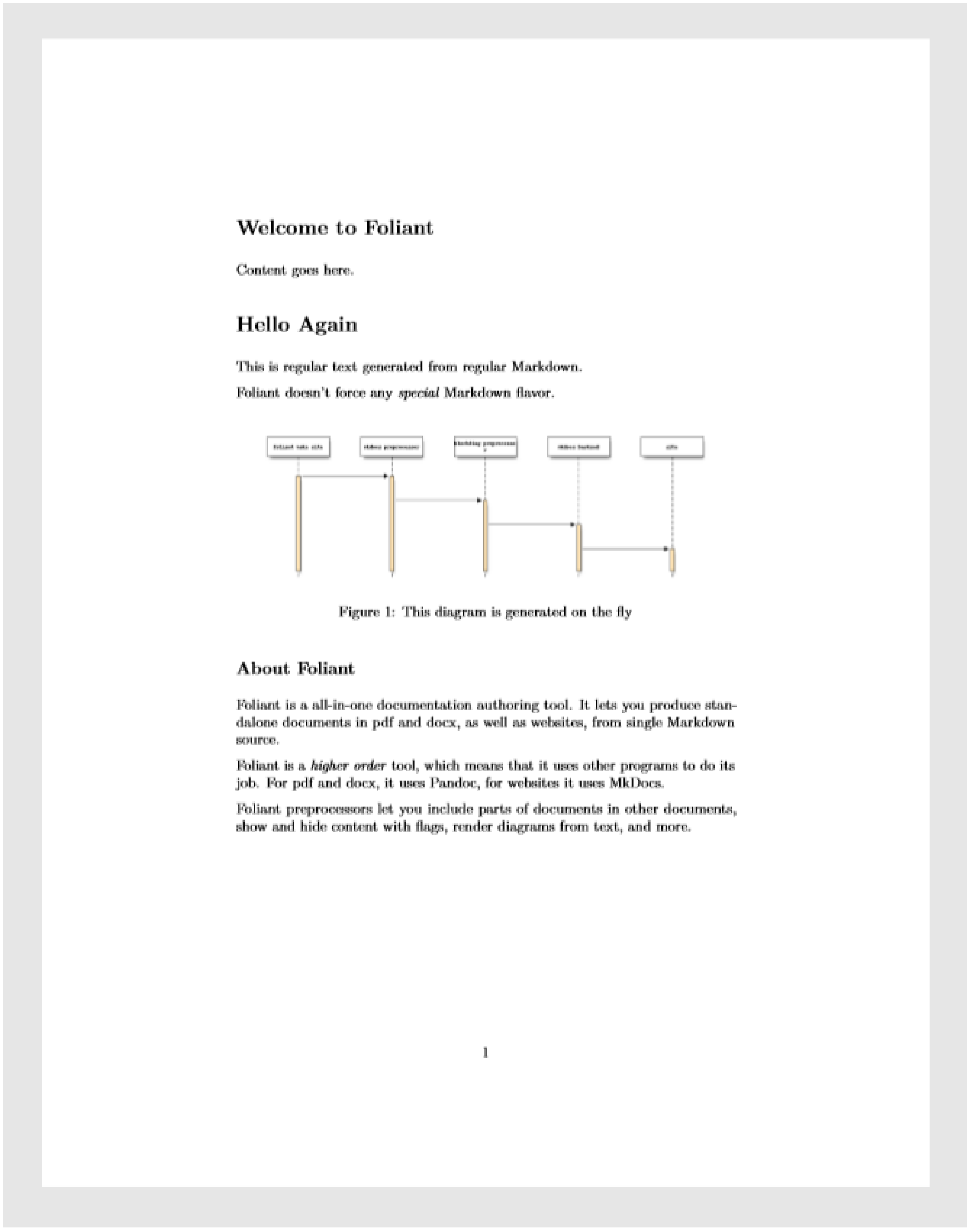
Foliant is a all-in-one documentation authoring tool. It lets you produce standalone documents in pdf and docx, as well as websites, from single Markdown source.

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.

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

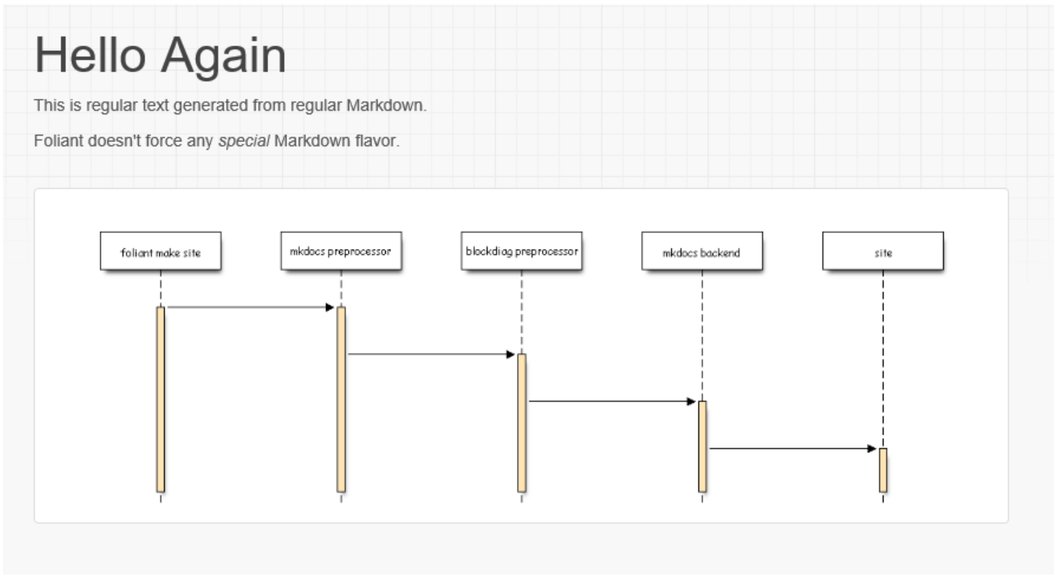
Figure 6. Sequence diagram drawn with seqdiag in the pdf

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:

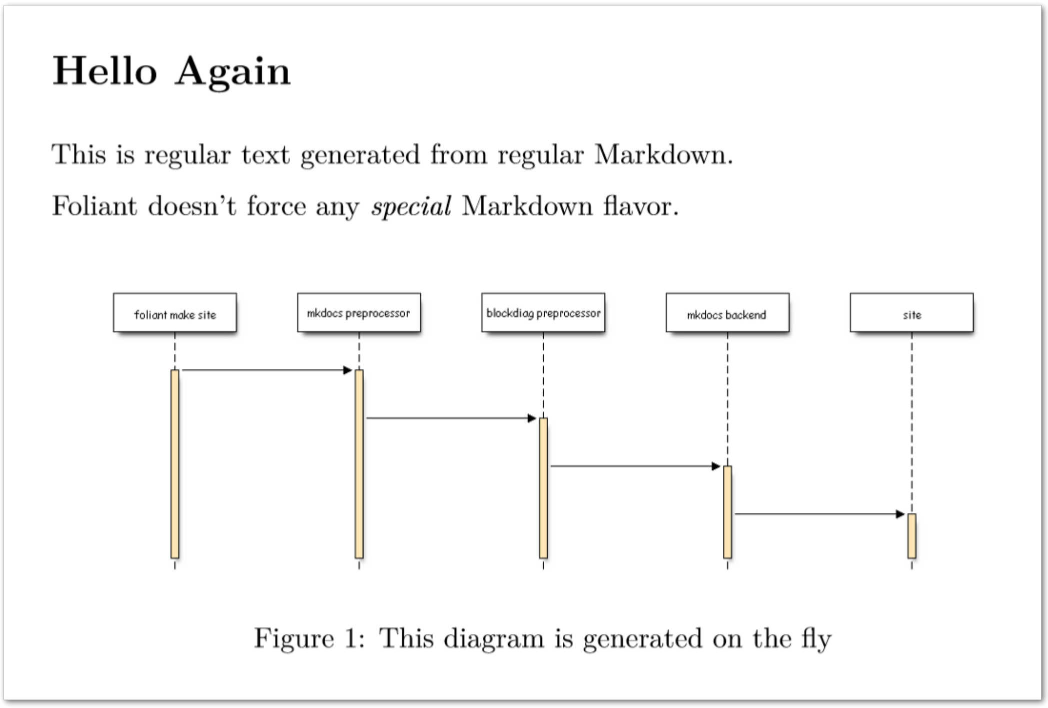


Hello Again

This is regular text generated from regular Markdown.

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

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



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

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="<if targets="html">png</if><if targets="pdf">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

Confluence

pypi v0.6.0

Confluence backend for Foliant

Confluence backend generates confluence articles and uploads them on your confluence server. It can create and edit pages in Confluence with content based on your Foliant project.

It also has a feature of restoring the user inline comments, added for the article, even after the commented fragment was changed.

This backend adds the `confluence` target for your Foliant `make` command.

Installation

```
$ pip install foliantcontrib.confluence
```

> The backend requires Pandoc to be installed in your system. Pandoc is needed to convert Markdown into HTML.

Usage

To upload a Foliant project to Confluence server use `make confluence` command:

```
$ foliant make confluence
Parsing config... Done
Making confluence... Done

Result:
https://my_confluence_server.org/pages/viewpage.action?pageId=123 (Page Title)
```

Config

You have to set up correct config for this backend to work properly.

Specify all options in `backend_config.confluence` section:

```yaml
backend_config:
  confluence:
    host: 'https://my_confluence_server.org'
    login: user
    password: user_password
    id: 124443
    title: Title of the page
    space_key: "~user"
    parent_id: 124442
    notify_watchers: false
    toc: false
    restore_comments: true
    resolve_if_changed: false
    pandoc_path: pandoc
```

`host` **Required** Host of your confluence server.

`login` Login of the user who has permissions to create and update pages. If login is not supplied, it will be prompted during build.

`password` Password of the user. If password is not supplied, it will be prompted during build.

`id` ID of the page where the content will be uploaded. *Only for already existing pages*

`title` Title of the page to be created or updated.

Remember that page titles in the space have to be unique.

`space_key` The space key where the page(s) will be created/edited. *Only for not yet existing pages.*

`parent_id` ID of the parent page under which the new one(s) should be created. *Only for not yet existing pages.*

`notify_watchers` If `true` — watchers will be notified that the page has changed. Default: `false`

`toc` Set to `true` to add table of contents to the beginning of the document. Default: `false`

`restore_comments` Attempt to restore inline comments near the same places after updating the page. Default: `true`

`resolve_if_changed` Delete inline comment from the source if the commented text was changed. This will automatically mark comment as resolved. Default: `false`

`pandoc_path` Path to Pandoc executable (Pandoc is used to convert Markdown into HTML).

User's guide

Uploading articles

By default if you specify `id` or `space_key` and `title` in foliant.yml, the whole project will be built and uploaded to this page.

If you wish to upload separate chapters into separate articles, you need to specify the respective `id` or `space_key` and `title` in *meta section* of the chapter.

Meta section is a YAML-formatted field-value section in the beginning of the document, which is defined like this:

```
---
field: value
field2: value
---

Your chapter md-content
```

If you want to upload a chapter into confluence, add its properties under the `confluence` key like this:

```
---
confluence:
    title: My confluence page
    space_key: "~user"
---
```

```

You chapter md-content
```

> **Important notice!** Both modes work together. If you specify the `id1` in foliant.yml and `id2` in chapter's meta — the whole project will be uploaded to the page with `id1`, and the specific chapter will also be uploaded to page with `id2`.

Creating pages

If you want a new page to be created for content in your Foliant project, just supply in foliant.yml the space key and a title which does not yet exist in this space. Remember that in Confluence page titles are unique inside one space. If you use a title of an already existing page, the backend will attempt to edit it and replace its content with your project.

Example config for this situation is:

```
backend_config:
  confluence:
    host: https://my_confluence_server.org
    login: user
    password: pass
    title: My unique title
    space_key: "~user"
```

Now if you change the title in your config, confluence will *create a new page with the new title*, leaving the old one intact.

If you want to change the title of your page, the answer is in the following section.

Updating pages

Generally to update the page contents you may use the same config you used to create it (see previous section). If the page with specified title exists, it will be updated.

Also, you can just specify the id of an existing page. After build its contents will be updated.

```
backend_config:
  confluence:
    host: https://my_confluence_server.org
    login: user
    password: pass
    id: 124443
```

This is also *the only* way to edit a page title. If `title` param is specified, the backend will attempt to change the page's title to the new one:

```
backend_config:
  confluence:
    host: https://my_confluence_server.org
    login: user
    password: pass
    id: 124443
    title: New unique title
```

Updating part of a page

Confluence backend can also upload an article into the middle of a Confluence page, leaving all the rest of it intact. To do this you need to add an *Anchor* into your page in the place where you want Foliant content to appear.

1. Go to Confluence web interface and open the article.
2. Go to Edit mode.
3. Put the cursor in the position where you want your Foliant content to be inserted and start typing `{anchor` to open the macros menu and locate the Anchor macro.
4. Add an anchor with the name `foliant`.
5. Save the page.

Now if you upload content into this page (see two previous sections), Confluence backend will leave all text which was before and after the anchor intact, and add your Foliant content in the middle.

You can also add two anchors: `foliant_start` and `foliant_end`. In this case all text between these anchors will be replaced by your Foliant content.

Inserting raw confluence tags

If you want to supplement your page with confluence macros or any other storage-specific html, you may do it by wrapping them in the `<raw_confluence></raw_confluence>` tag.

For example, if you wish to add a table of contents into the middle of the document for some reason, you can do something like this:

```
Lorem ipsum dolor sit amet, consectetur adipisicing elit. Odit dolorem nulla quam doloribus delectus voluptate.

<raw_confluence><ac:structured-macro ac:macro-id="1" ac:name="toc" ac:schema-version="1"/></raw_confluence>

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Officiis, laboriosam cumque soluta sequi blanditiis, voluptatibus quaerat similique nihil debitis repellendus.
```

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:
    mkdocs_path: mkdocs
    slug: my_awesome_project
    use_title: true
    use_chapters: true
    use_headings: true
    default_subsection_title: Expand
    mkdocs.yml:
      site_name: Custom Title
      site_url: http://example.com
      site_author: John Smith
```

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

`slug` Result directory name without suffix (e.g. `.mkdocs`). Overrides top-level config option `slug`.

`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`.

`use_headings` If `true`, the resulting data of `pages` section in `mkdocs.yml` will be updated with the content of top-level headings of source Markdown files.

`default_subsection_title` Default title of a subsection, i.e. a group of nested chapters, in case the title is specified as an empty string. If `default_subsection_title` is not set in the config, **…** will be used.

`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:
      ...
    slug: My_Awesome_Custom_Slug
```

`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.

`slug` is the result file name without suffix (e.g. `.pdf`). Overrides top-level config option `slug`.

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`.

LaTeX listings package does not work correctly with non-ASCII characters, e.g. Cyrillic letters

If you use non-ASCII characters inside backticks in your document, you can see an error like this:

```
Error producing PDF.
! Undefined control sequence.
\lst@arg ->git clone [SSHк-
                              люч]
l.627 ...through{\lstinline!git clone [SSHключ-]!}
```

To fix it, set `listings` parameter to `false`:

```
backend_config:
  pandoc:
    ...
    params:
      pdf_engine: xelatex
      listings: false
      number_sections: true
    ...
```

Slate

Slate backend generates API documentation from Markdown using Slate docs generator.

This backend operates two targets:

- `site` — build a standalone website;
- `slate` — generate a slate project out from your Foliant project.

Installation

```
$ pip install foliantcontrib.slate
```

Usage

To use this backend Slate should be installed in your system. Follow the instruction in Slate repo.

To test if you've installed Slate properly head to the cloned Slate repo in your terminal and try the command below. You should get similar response.

```
$ bundle exec middleman
== The Middleman is loading
== View your site at ...
== Inspect your site configuration at ...
```

To convert Foliant project to Slate:

```
$ foliant make slate✔
 Parsing config✔
 Making slate─────────────────────

Result: My_Project-2018-09-19.src/
```

Build a standalone website:

```
$ foliant make site✔
 Parsing config✔
 Making site─────────────────────

Result: My_Project-2018-09-19.slate/
```

Config

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

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

```
backend_config:
  slate:
    shards: data/shards
    header:
        title: My API documentation
        language_tabs:
          - xml: Response example
        search: true
```

`shards` Path to the shards directory relative to Foliant project dir or list of such paths. Shards allow you to customize Slate's layout, add scripts etc. More info on shards in the following section. Default: `shards`

`header` Params to be copied into the beginning of Slate main Markdown file `index.html.md`. They allow you to change the title of the website, toggle search and add language tabs. More info in Slate Wiki.

About shards

Shards is just a folder with files which will be copied into the generated Slate project replacing all files in there. If you follow the Slate project structure you can replace stylesheets, scripts, images, layouts etc to customize the view of the resulting site.

If shards is a string — it is considered a path to single shards directory relative to Foliant project dir:

```
slate:
    shards: 'data/shards'
```

If shards is a list — each list item is considered as a shards dir. They will be copied into the Slate project subsequently with replace.

```
slate:
    shards:
        - 'common/shards'
        - 'custom/shards'
        - 'new_design'
```

For example, I want to customize standard Slate stylesheets. I look at the Slate repo and see that they lie in the folder `<slate>/source/stylesheets`. I create new stylesheets with the same names as the original ones and put them into my shards dir like that:

```
shards\
    source\
        stylesheets\
            _variables.scss
            screen.css.scss
```

These stylesheets will replace the original ones in the Slate project just before the website is be baked. So the page will have my styles in the end.

Preprocessors

General Notes

Most simple preprocessors apply unconditionally to the whole content of each Markdown file in the Foliant project. But usually preprocessors look for some specific pseudo-XML tags in Markdown content. Each preprocessor registers its own set of tags.

Tags can have attributes and a body. Attributes are usually used to specify some required or optional parameters. Body is the content that is enclosed between opening and closing tags; preprocessors usually do something with this content:

```
<tag attribute_1="value_1" ... attribute_N="value_N">body</tag>
```

Foliant under 1.0.8 tries to convert each attribute value into a boolean value, a number, or a string. Attribute values must be enclosed in double quotes (").

Since Foliant 1.0.9, attribute values are processed as YAML. Scalar values are also converted into boolean values, numbers and strings, but you may specify composite values that should be transformed into lists or dictionaries. You may also use modifiers (i.e. YAML tags) that are available in the Foliant project's config.

`!path` The string preceded by this modifier should be converted into an existing path relative to the Foliant project's top-level ("root") directory.

`!project_path` The string preceded by this modifier should be converted into a path relative to the Foliant project's top-level ("root") directory. This path may be nonexistent.

`!rel_path` The string preceded by this modifier should be converted into a path relative to the currently processed Markdown file. This path may be nonexistent.

If you develop a preprocessor that accepts some path, by default it is better to be a path relative to the currently processed Markdown file.

Also, since Foliant 1.0.9, attribute values may be enclosed into double (") or single (') quotes.

Admonitions

pypi v1.0.0

Admonitions preprocessor for Foliant

Preprocessor which tries to make admonitions syntax available for most backends.

Admonitions are decorated fragments of text which indicate a warning, notice, tip, etc.

We use rST-style syntax for admonitions which is already supported by mkdocs backend with `admonition` extension turned on. This preprocessor makes this syntax work for pandoc and slate backends.

Installation

```
$ pip install foliantcontrib.admonitions
```

Config

Just add `admonitions` into your preprocessors list. Right now the preprocessor doesn't have any options:

```
preprocessors:
    - admonitions
```

Usage

Add an admonition to your Markdown file:

```
!!! warning "optional admonition title"
    Admonition text.

    May be several paragraphs.
```

Notes for slate

Slate has its own admonitions syntax of three types: `notice` (blue notes), `warning` (red warnings) and `success` (green notes). If another type is supplied, slate draws a blue note but without the “i” icon.

Admonitions preprocessor transforms some of the general admonition types into slate’s for convenience (so you could use `error` type to display same kind of note in both slate and mkdocs). These translations are indicated in the table below:

original type	translates to
error	warning
danger	warning
caution	warning
info	notice
note	notice
tip	notice
hint	notice

Anchors

pypi v1.0.2

Anchors

Preprocessor which allows to use arbitrary anchors in Foliant documents.

Installation

```
$ pip install foliantcontrib.anchors
```

Config

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

```
preprocessors:
    - anchors
```

The preprocessor has some options, but most probably you won't need any of them:

```
preprocessors:
    - anchors:
        element: '<span id="{anchor}"></span>'
        tex: False
```

`element` Template of an HTML-element which will be placed instead of the `<anchor>` tag. In this template `{anchor}` will be replaced with the tag contents. Default: `'<span id="{anchor}"></span>'`

`tex` If this option is `True`, preprocessor will try to use TeX-language anchors: `\hypertarget{anchor}{}`. Default: `False`

Notice, this option will work only with `pdf` target. For all other targets it is set to `False`.

Usage

Just add an `anchor` tag to some place and then use an ordinary Markdown-link to this anchor:

```
...

<anchor>limitation</anchor>
Some important notice about system limitation.

...

Don't forget about [limitation](#limitation)!
```

You can also place anchors in the middle of paragraph:

```

Lorem ipsum dolor sit amet, consectetur adipisicing elit.<
anchor>middle</anchor> Molestiae illum iusto, sequi magnam
```

```
consequatur porro iste facere at fugiat est corrupti dolorum
 quidem sapiente pariatur rem, alias unde! Iste, aliquam.

[Go to the middle of the paragraph](#middle)
```

You can place anchors inside tables:

```

Name | Age | Weight
---- | --- | ------
Max  | 17  | 60
Jane | 98  | 12
John | 10  | 40
Katy | 54  | 54
Mike <anchor>Mike</anchor>| 22  | 299
Cinty| 25  | 42

...

Something's wrong with Mike, [look](#Mike)!
```

Additional info

1. Anchors are case sensitive

`Markdown` and `MarkDown` are two different anchors.

2. Anchors should be unique

You can’t use two anchors with the same name in one document.

If preprocessor notices repeating anchors in one md-file it will throw you a warning.

If there are repeating anchors in different md-files and they all go into single pdf or docx, all links will lead to the first one.

3. Anchors may conflict with headers

Headers are usually assigned anchors of their own. Be careful, your anchors may conflict with them.

Preprocessor will try to detect if you are using anchor which is already taken by the header and warn you in console.

Remember, that header anchors are almost always in lower-case and almost never use special symbols except `-`.

4. Some symbols are restricted

You can’t use these symbols in anchors:

```
[]<>\"
```

Also you can’t use space.

5. But a lot of other symbols are available

All these are valid anchors:

```
<anchor>!important!</anchor>
<anchor>_anchor_</anchor>
<anchor>section(1)</anchor>
<anchor>section/1/</anchor>
<anchor>anchor$1$</anchor>
<anchor>about:info</anchor>
<anchor>test'1';</anchor>
<anchorякорь></anchor>
<anchor□></anchor>
```

Notice for Mkdocs

In many Mkdocs themes the top menu lays over the text with absolute position. In this situation all anchors will be hidden by the menu.

Possible solution is to use `element` option. Example config:

```
preprocessors:
    - anchors:
        element: '<span style="display:block; margin:-3.1rem; padding:3.1rem;" id="{anchor}"></span>'
```

APILinks

Preprocessor for replacing API *reference*s in markdown files with links to actual method description on the API documentation web-page.

Installation

```
$ pip install foliantcontrib.apilinks
```

Quick Start

Say, you have an API documentation hosted at the url http://example.com/api-docs

On this page you have HTML headings before each method description which look like this:

```
<h2 id="get-user-authenticate">GET user/authenticate</h2>
```

You want references to these methods in your documentation to be replaced with the links to the actual method descriptions. Your references look like this:

```
To authenticate user use API method `GET user/authenticate`.
```

Now all you need to do is add the apilinks preprocessor into your foliant.yml and state your API url in its options like this:

```
preprocessors:
    - apilinks:
        API:
            My-API:
                url: http://example.com/api-docs
```

Here:

- `API` is a required section;
- `My-API` is a local name of your API. Right now it is not very important but will come in handy in the next example;

– `url` is a string with full url to your API documentation web-page. It will be used to validate references and to construct a link to method.

After foliant applies the preprocessor your document will be transformed into this:

```
To authenticate user use API method [GET user/authenticate](
http://example.com/api-docs/#get-user-authenticate).
```

Notice that preprocessor figured out the correct anchor `#get-user-authenticate` by himself. Now instead of plain name of the method you've got a link to the method description!

Ok, what if I have two different APIs: client API and admin API?

No problem, put both of them into your config:

```
preprocessors:
    - apilinks:
        API:
            Client-API:
                url: http://example.com/client/api-docs
            Admin-API:
                url: http://example.com/admin/api-docs
```

Now this source:

```
To authenticate user use API method `GET user/authenticate`.
To ban user from the website use admin API method `POST
admin/ban_user/{user_id}`
```

Will be transformed by apilinks into this:

```
To authenticate user use API method [GET user/authenticate](
http://example.com/client/api-docs/#get-user-authenticate).
To ban user from the website use admin API method [POST
admin/ban_user/{user_id}](http://example.com/admin/api-docs
/#post-admin-ban_user-user_id)
```

Notice that apilinks determined that the first reference is from Client API, and the second one is from the Admin API. How is that possible? Easy: preprocessor parses each API url from the config and stores their methods before looking for references. When the time comes to process the references it already has a list of all methods to validate your reference and to determine which API link should be inserted.

But what if we have the same-named method in both of our APIs? In this case you will see a warning:

```
WARNING: GET /service/healthcheck is present in several APIs
 (Client-API, Admin-API). Please, use prefix. Skipping
```

It suggests us to use prefix, and by that it means to prefix the reference by the local name of the API in config. Like that:

```
Check status of the server through Client API: `Client-API: GET /service/healthcheck`
Do the same through Admin API: `Admin-API: GET /service/healthcheck`
```

Here `Client-API:` and `Admin-API:` are prefixes. And they should be the same as your API names in the config.

Now each reference will be replaced with the link to corresponding API web-page.

---

apilinks is a highly customizable preprocessor. You can tune:

- the format of the references;
- the output string which will replace the reference;
- the format of the headings in your API web-page;
- and more!

For details look through the following sections.

Glossary:

- **reference** — reference to an API method in the source file. The one to be replaced with the link, e.g. `GET user/config`
- **verb** — HTTP method, e.g. `GET`, `POST`, etc.

- **command** — resource used to represent method on the API documentation web-page, e.g. `/service/healthcheck`.
- **endpoint prefix** — A prefix from server root to the command. If the command is `/user/status` and full resource is `/api/v0/user/satus` then the endpoint-prefix should be stated `/api/v0`. In references you can use either full resource (`{endpoint_prefix}/{command}`) or just the command. apilinks will sort it out for you.
- **output** — string, which will replace the *reference*.
- **header** — HTML header on the API documentation web-page of the method description, e.g. `<h2 id="get-user-config">GET user/config</h2>`
- **anchor** — web-anchor leading to the specific *header* on the API documentation web-page, e.g. `#get-user-config`

How Does It Work?

Preprocessor can work in *online* and *offline* modes.

In offline mode it merely replaces *references* to API methods with links to their description. The references are catched by a regular expression. The link url is taken from config and the link *anchor* is generated from the reference automatically.

You can have several different APIs stated in the config. You can use prefixes to point out which API is being *reference*d. Prefixes format may be customized in the configuration but by default you do it like this: `Client-API: GET user/name`. Here "*Client-API*" is a prefix.

If you don't use prefix in the *reference* preprocessor will suppose that you meant the default API, which is marked by `default` option in config. If none of them is marked — goes for the first in list.

In online mode things are getting interesting. Preprocessor actually goes to each of the API web-pages, and collects all method **headers** (right now only `h2` headers are supported). Then it goes through your document's source: when it meets a *reference*, it looks through all the collected methods and replaces the reference with the correct link to it. If method is not found — preprocessor will show warning and leave the reference unchanged. Same will happen if there are several methods with this name in different APIs.

Prefixes, explained before, are supported too.

Config

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

```yaml
preprocessors:
  - apilinks
```

The preprocessor has a lot of options. For your convenience the required options are marked *(required)*; and those options which are used in customization are marked *(optional)*. Most likely you will need just one or two of the latter.

```yaml
preprocessors:
- apilinks:
    targets:
        - site
    offline: False
    trim_if_targets:
        - pdf
    prefix_to_ignore: Ignore
    reference:
        - regex: *ref_pattern
          only_with_prefixes: false
          only_defined_prefixes: true
          output_template: '[{verb} {command}]({url})'
          trim_template: '`{verb} {command}`'
    API:
        Client-API:
            url: http://example.com/api/client
            default: true
            header_template: '{verb} {command}'
        Admin-API:
            url: http://example.com/api/client
            header_template: '{command}'
            endpoint-prefix: /api/v0
```

`prefix_to_ignore` *(optional)* A default prefix for ignoring references. If apilinks meets a reference with this prefix it leaves it unchanged. Default: `Ignore`

`targets` *(optional)* List of supported targets for `foliant make` command. If target is not listed here — preprocessor won't be applied. If the list is empty — preprocessor will be applied for any target. Default: `[]`

`offline` *(optional)* Option determining whether the preprocessor will work in *online* or *offline* mode. Details in the **How Does It Work?** and **Online and Offline Modes Comparison** sections. Default: `False`

`trim_if_targets` *(optional)* List of targets for `foliant make` command for which the prefixes from all *references* in the text will be cut out. Default: `[]`

Only those references whose prefixes are defined in the `API` section (described below) are affected by this option. All references with unlisted prefixes will not be trimmed.

`reference` *(optional)* A subsection for listing all the types of references you are going to catch in the text, and their properties. Options for this section are listed below.

Reference options `regex` : *(optional)* regular expression used to catch *references* in the source. Look for details in the **Capturing References** section. Default:

```
(?P<source>`((?P<prefix>[\w-]+):\s*)?(?P<verb>OPTIONS|GET|
HEAD|POST|PUT|DELETE|TRACE|CONNECT|PATCH|LINK|UNLINK)\s+(?P<
command>\S+)`)
```

`only_with_prefixes` *(optional)* if this is `true`, only *references* with prefix will be transformed. Ordinary links like `GET user/info` will be ignored. Default: `false`

`only_defined_prefixes` *(optional)* if this is `true` all references whose prefix is not listed in the `API` section (described below) will be ignored, left unchanged. References without prefix are not affected by this option. Default: `false`.

`output_template` *(optional)* A template string describing the *output* which will replace the *reference*. More info in the **Customizing Output** section. Default: `'[{verb} {command}]({url})'`

`trim_template` *(optional)* Only for targets listed in `trim_if_targets` option. Tune this template if you want to customize how apilinks cuts out prefixes. The reference will be replaced with text based on this template. Default: ``'`{verb} {command}`'``

`API` *(required)* A subsection for listing all the APIs and their properties. Under this section there should be a separate subsection for each API. The section name represents the API name and, at the same time, the *prefix* used in the references. You need to add at least one API subsection for preprocessor to work.

API properties

`url` *(required)* An API documentation web-page URL. It will be used to construct the full link to the method. In online mode it will also be parsed by preprocessor for validation.

`default` *(optional)* Only for offline mode. Marker to define the default API. If several APIs are marked default, preprocessor will choose the first of them. If none is marked default — the first API in the list will be chosen. The value of this item should be `true`.

`header_template` *(optional)* A template string describing the format of the headings in the API documentation web-page. Details in **parsing API web-page** section. Default: `'{verb} {command}'`

`endpoint-prefix` *(optional)* The endpoint prefix from the server root to API methods. If is stated — apilinks can divide the command in the reference and search for it more accurately. Also you could use it in templates. More info coming soon. Default: `''`

Online and Offline Modes Comparison

Let's study an example and look how the behavior of the preprocessor will change in online and offline modes.

We have three APIs described in the config:

```
preprocessors:
  - apilinks:
      API:
        Admin-API:
            url: http://example.com/api/client
        Client-API:
            url: http://example.com/api/client
            default: true
            header_template: '{verb} {command}'
        Remote-API:
            url: https://remote.net/api-ref/
            header_template: '{command}'
```

Now let's look at different examples of the text used in Markdown source and how it is going to be transformed in Offline and Online modes

Example 1 Source:

```
Unprefixed link which only exists in Remote API: `GET system
/info`.
```

In *Offline mode* preprocessor won't do any checks and just replace the reference with the link to default API from the config:

```
Unprefixed link which only exists in Remote API: [GET system
/info](http://example.com/api/client/#get-system-info).
```

This is certainly a wrong decision, but it is our fault, we sould have added the prefix to the reference.

But let's look what will happen in *Online mode*:

```
Unprefixed link which only exists in Remote API: [GET system
/info](https://remote.net/api-ref/#system-info).
```

Without any prefix the preprocessor determined that it should choose the Remote API to replace this reference because this method exists only on its page. The `default` option is just ignored in this mode.

By the way, notice how anchors differ in the two examples. For Remote API preprocessor used its header template to reconstruct the anchor, dropping the verb from it.

Example 2 Source:

```
Unprefixed link with misprint: `GET user/sttus`.
The link is incorrect, there's no such method in any of the APIs.
```

In *Offline mode* preprocessor won't do any checks again. No magic, the reference will be replaced with the link to default API from the config:

```
Unprefixed link with misprint: [GET user/sttus](http://example.com/api/client/#get-user-sttus).
The link is incorrect, there's no such method in any of the APIs.
```

In *Online mode* preprocessor won't be able to find the method during validation and the reference won't be replaced at all:

```
Unprefixed link with misprint: `GET user/sttus`.
The link is incorrect, there's no such method in any of the APIs.
```

During the Foliant project assembly you will see a warning message:

```
WARNING: Cannot find method GET user/sttus. Skipping
```

Example 3 Source:

```
Prefixed link to the Admin API: `Admin-API: POST user/
ban_forever`.
```

In *Offline mode* preprocessor will notice the prefix and will be able to replace the reference with an appropriate link:

```
Prefixed link to the Admin API: [POST user/ban_forever](http
://example.com/api/client/#post-user-ban_forever).
```

Notice that prefix disappeared from the text. If you wish it to stay there — edit the `output_template` option to something like this: `'{prefix}: {verb} {command}'`.

In *Online mode* the result will be exactly the same. Preprocessor will check the Admin-API methods, find there the referenced method and replace it in the text:

```
Prefixed link to the Admin API: [POST user/ban_forever](http
://example.com/api/client/#post-user-ban_forever).
```

Example 4

```
Prefixed link to the Remote API with a misprint: `Remote-API
: GET billling/info`.
Oh no, the method is incorrect again.
```

In *Offline mode* preprocessor will perform no checks and just replace the reference with the link to Remote API:

```
Prefixed link to the Remote API with a misprint: [GET
billling/info](https://remote.net/api-ref/#get-billling-info
).
Oh no, the method is incorrect again.
```

Online mode, on the other hand, will make its homework. It will check whether the Remote API actually has the method *GET billling/info*. Finding out that it hasn't it will leave the reference unchanged:

```
Prefixed link to the Remote API with a misprint: `Remote-API
: GET billling/info`.
Oh no, the method is incorrect again.
```

…and warn us with the message:

```
WARNING: Cannot find method GET billling/info in Remote-API.
 Skipping
```

Example 5

```
Now let's reference a method which is present in both Client
 and Admin APIs: `GET service/healthcheck`.
```

In *Offline mode* preprocessor will just replace the reference with a link to default API:

```
Now let's reference a method which is present in both Client
 and Admin APIs: [GET service/healthcheck](http://example.
com/api/client/#get-service-healthcheck).
```

But in *Online mode* preprocessor will go through all API method lists. It will find several mentions of this exact method and, confused, won't replace the reference at all:

```
Now let's reference a method which is present in both Client
 and Admin APIs: `GET service/healthcheck`.
```

You will also see a warning:

```
WARNING: GET /service/healthcheck is present in several APIs
 (Admin-API, Client-API). Please, use prefix. Skipping
```

Capturing References

apilinks uses regular expressions to capture *references* to API methods in Markdown files.

The default reg-ex is as following:

```
(?P<source>`((?P<prefix>[\w-]+):\s*)?(?P<verb>OPTIONS|GET|
HEAD|POST|PUT|DELETE|TRACE|CONNECT|PATCH|LINK|UNLINK)\s+(?P<
command>\S+)`)
```

This expression accepts references like these:

- `Client-API: GET user/info`
- `UPDATE user/details`

> Notice that default expression uses *Named Capturing Groups*. You would probably want to use all of them too if you are to redefine the expression. Though not all of them are required, see the table below.

Group	Required	Description
source	YES	The full original reference string
prefix	NO	Prefix pointing to the name of the API from config
verb	NO	HTTP verb as `GET`, `POST`, etc
command	YES	the full method resource as it is stated in the API header (may include endpoint prefix)

To redefine the regular expression add an option `reg-regex` to the preprocessor config.

For example, if you want to capture ONLY references with prefixes you may use the following:

```
preprocessors:
  - apilinks:
      reference:
```

```
      - regex: '(?P<source>`((?P<prefix>[\w-]+):\s*)(?P<verb
>POST|GET|PUT|UPDATE|DELETE)\s+(?P<command>\S+)`)'
```

> This example is for illustrative purposes only. You can achieve the same goal by just switching on the `only_with_prefixes` option.

Now the references without prefix (`UPDATE user/details`) will be ignored.

Customizing Output

You can customize the *output*-string which will replace the *reference* string. To do that add a template into your config-file.

A *template* is a string which may contain properties, surrounded by curly braces. These properties will be replaced with the values, and all the rest will remain unchanged.

For example, look at the default template:

```
preprocessors:
  - apilinks:
    reference:
      - output_template: '[{verb} {command}]({url})',
```

> Don't forget the single quotes around the template. This way we say to yaml engine that this is a string for it not to be confused with curly braces.

With the default template, the reference string will be replaced by something like that:

```
[GET user/info](http://example.com/api/#get-user-info)
```

If you don't want references to be transfromed into links, use your own template. Properties you may use in the template:

property	description	example
url	Full url to the method description	`http://example.com/api/#get-user-info`

property	description	example
source	Full original reference string	‘`Client-API: GET user/info`‘
prefix	Prefix used in the reference	`Client-API`
verb	HTTP verb used in the reference	`GET`
command	API command being referenced with endpoint prefix removed	`user/info`
endpoint_prefix	Endpoint prefix to the API (if `endpoint-prefix` option is filled in)	`/api/v0`

Parsing API Web-page

apilinks goes through the API web-page content and gathers all the methods which are described there.

To do this preprocessor scans each HTML `h2` tag and stores its `id` attribute (which is an *anchor* of the link to be constructed) and the contents of the tag (the *heading* itself).

For example in this link:

```
<h2 id="get-user-info">GET user/info</h2>
```

the anchor would be `get-user-info` and the heading would be `GET user/info`.

To construct the link to the method description we will have to create the correct anchor for it. To create an anchor we would need to reconstruct the heading first. But the heading format may be arbitrary and that's why we need the `header_template` config option.

The `header_template` is a string which may contain properties, surrounded by curly braces. These properties will be replaced with the values, when preprocessor will attempt to reconstruct the heading. All the rest will remain unchanged.

For example, if your API headings look like this:

```
<h2 id="method-user-info-get">Method user/info (GET)</h2>
```

You should use the following option:

```
...
API:
    Client-API:
        header_template: 'Method {command} ({verb})'
...
```

> Don't forget the single quotes around the template. This way we say to yaml engine that this is a string.

If your headers do not have a verb at all:

```
<h2 id="user-info">user/info</h2>
```

You should use the following option:

```
...
API:
    Client-API:
        header_template: '{command}'
...
```

Properties you may use in the template:

property	description	example
verb	HTTP verb used in the reference	`GET`
command	API command being referenced	`user/info`
endpoint_prefix	Endpoint prefix to the API (if `endpoint-prefix` option is filled in)	`/api/v0`

Badges

pypi v1.0.2

Badges

Preprocessor for Foliant which helps to add badges to your documents. It uses Shields.io to generate badges.

Installation

```
$ pip install foliantcontrib.badges
```

Config

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

```
preprocessors:
    - badges
```

The preprocessor has a number of options:

```
preprocessors:
    - badges:
        server: 'https://img.shields.io'
        as_object: true
        add_link: true
        vars:
            jira_path: localhost:3000/jira
            package: foliant
        # badge look parameters
        style: flat-square
        logo: jira
```

`server` Shields server URL, which hosts badges. default: `https://img.shields.io`

`as_object` If `true` — preprocessor inserts `svg` badges with HTML `<object>` tag, instead of Markdown image tag. This is required for links and hints to work. default: `true`

`add_link` If `true` preprocessor tries to determine the link which should be added to badge (for example, link to jira issue page for jira issue badge). Only works with `as_object = true`. default: `true`

> Please note that right now only links for **pypi** and **jira-issue** badges are being added automatically. Please contribute or contact author for adding other services.

`vars` Dictionary with variables which will be replaced in badge urls. See **variables** section.

Also you may add parameters specified on the shields.io website which alter the badge view like: `label`, `logo`, `style` etc.

Usage

Just add the `badge` tag and specify path to badge in the tag body:

```
<badge>jira/issue/https/issues.apache.org/jira/kafka-2896.
svg</badge>
```

All options from config may be overriden in tag parameters:

```
<badge style="social" as_object="false">jira/issue/https/
issues.apache.org/jira/kafka-2896.svg</badge>
```

Variables

You can use variables in your badges to replace parts which repeat often. For example, if we need to add many badges to our Jira tracker, we may put the protocol and host parameters into a variable like this:

```
preprocessors:
    - badges:
        vars:
            jira: https/issues.apache.org/jira
```

To reference a variable in a badge path use syntax `${variable}`:

```
<badge>jira/issue/${jira}/kafka-2896.svg</badge>

Description of the issue goes here. But it's not the only one.

<badge>jira/issue/${jira}/KAFKA-7951.svg</badge>

Description of the second issue.
```

BindSympli

BindSympli is a tool to download design layout images from Sympli CDN using certain Sympli account, to resize these images, and to bind them with the documentation project.

Installation

Before using BindSympli, you need to install Node.js, Puppeteer, wget, and ImageMagick.

BindSympli preprocessor code is written in Python, but it uses the external script written in JavaScript. This script is provided in BindSympli package:

```
$ pip install foliantcontrib.bindsympli
```

Config

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

```
preprocessors:
    - bindsympli
```

The preprocessor has a number of options with the following default values:

```
preprocessors:
    - bindsympli:
        get_sympli_img_urls_path: get_sympli_img_urls.js
        wget_path: wget
        convert_path: convert
        cache_dir: !path .bindsymplicache
        sympli_login: ''
        sympli_password: ''
        image_width: 800
        max_attempts: 5
```

`get_sympli_img_urls_path` Path to the script `get_sympli_img_urls.js` or alternative command that launches it (e.g. `node some_another_script.js`). By default, it is assumed that you have this command and all other commands in PATH.

`wget_path` Path to wget binary.

`convert_path` Path to `convert` binary, a part of ImageMagick.

`cache_dir` Directory to store downloaded and resized images.

`sympli_login` Your username in Sympli account.

`sympli_password` Your password in Sympli account.

`image_width` Width of resulting images in pixels (original images are too large).

`max_attempts` Maximum number of attempts to run the script `get_sympli_img_urls.js` on fails.

Usage

To insert a design layout image from Sympli into your documentation, use `<<sympli>...</sympli>` tags in Markdown source:

```
’
Heres an image from Sympli:
```

```

<<sympli caption="An optional caption" width="400" url="
https://app.sympli.io/app#!/designs/0123456789abcdef01234567
/specs/assets"></sympli>
```

You have to specify the URL of Sympli design layout page in `url` attribute.

You may specify an optional caption in the `caption` attribute, and an optional custom image width in the `width` attribute. The `width` attribute overrides the `image_width` config option for a certain image.

BindSympli preprocessor will replace such blocks with local image references.

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>
```

CSVTables

This preprocessor converts csv data to markdown tables.

Installation

```
$ pip install foliantcontrib.csvtables
```

Config

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

```
preprocessors:
  - csvtables
```

The preprocessor has a number of options (default values stated below):

```
preprocessors:
    - csvtables:
        delimiter: ';'
        padding_symbol: ' '
        paddings_number: 1
```

`delimiter` Delimiter of csv data.

`padding_symbol` Symbol combination that will be places around datum (reversed on the right side).

`paddings_number` Symbol combination multiplier.

Usage

You can place csv data in `csvtable` tag.

```
<csvtable>
    Header 1;Header 2;Header 3;Header 4;Header 5
    Datum 1;Datum 2;Datum 3;Datum 4;Datum 5
    Datum 6;Datum 7;Datum 8;Datum 9;Datum 10
</csvtable>
```

Or in external `file.csv`.

```
<csvtable src="table.csv"></csvtable>
```

You can reassign setting for certain csv tables.

```
<csvtable delimiter=":" padding_symbol=" *">
    Header 1:Header 2:Header 3:Header 4:Header 5
    Datum 1:Datum 2:Datum 3:Datum 4:Datum 5
    Datum 6:Datum 7:Datum 8:Datum 9:Datum 10
</csvtable>
```

Example

`Usage` section will be converted to this:

You can place csv data in `csvtable` tag.

```
| Header 1 | Header 2 | Header 3 | Header 4 | Header 5 |
|----------|----------|----------|----------|----------|
| Datum 1  | Datum 2  | Datum 3  | Datum 4  | Datum 5  |
| Datum 6  | Datum 7  | Datum 8  | Datum 9  | Datum 10 |
```

Or in external `file.csv`.

```
| Header 1 | Header 2 | Header 3 | Header 4 | Header 5 |
|----------|----------|----------|----------|----------|
| Datum 1  | Datum 2  | Datum 3  | Datum 4  | Datum 5  |
| Datum 6  | Datum 7  | Datum 8  | Datum 9  | Datum 10 |
```

You can reassign setting for certain csv tables.

```
| *Header 1* | *Header 2* | *Header 3* | *Header 4* | *
Header 5* |
|------------|------------|------------|------------|------------|

| *Datum 1*  | *Datum 2*  | *Datum 3*  | *Datum 4*  | *Datum
 5*  |
| *Datum 6*  | *Datum 7*  | *Datum 8*  | *Datum 9*  | *Datum
 10* |
```

CustomIDs

CustomIDs is a preprocessor that allows to define custom identifiers (IDs) for headings in Markdown source by using Pandoc-style syntax in projects built with MkDocs or another backend that provides HTML output. These IDs may be used in hyperlinks that refer to a specific part of a page.

Installation

```
$ pip install foliantcontrib.customids
```

Usage

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

```
preprocessors:
    - customids
```

The preprocessor supports the following options:

```
    - customids:
        stylesheet_path: !path customids.css
        targets:
            - pre
            - mkdocs
            - site
            - ghp
```

`stylesheet_path` Path to the CSS stylesheet file. This stylesheet should define rules for `.custom_id_anchor_container`, `.custom_id_anchor_container_level_N`, `.custom_id_anchor`, and `.custom_id_anchor_level_N` classes. Here `N` is the heading level (`1` to `6`). Default path is `customids.css`. If stylesheet file does not exist, default built-in stylesheet will be used.

`targets` Allowed targets for the preprocessor. If not specified (by default), the preprocessor applies to all targets.

Custom ID may be specified after a heading content at the same line. Examples of Markdown syntax:

```
### First Heading {#custom_id_for_first_heading}

A paragraph.

#### Second Heading {#custom_id_for_second_heading}

Some another paragraph.
```

This Markdown source will be finally transformed into the HTML code:

```
<div class="custom_id_anchor_container
custom_id_anchor_container_level_1"><div id="
custom_id_for_first_heading" class="custom_id_anchor
custom_id_anchor_level_1"></div></div>
```

```

<h1>First Heading</h1>

<p>A paragraph.</p>

<div class="custom_id_anchor_container custom_id_anchor_container_level_2"><div id="custom_id_for_second_heading" class="custom_id_anchor custom_id_anchor_level_2"></div></div>

<h2>Second Heading</h2>

<p>Some another paragraph.</p>
```

(Note that CustomIDs preprocessor does not convert Markdown syntax into HTML; it only inserts HTML tags `<div class="custom_id_anchor_container">...</div>` into Markdown code.)

Custom IDs must not contain spaces and non-ASCII characters.

Examples of hyperlinks that refer to custom IDs:

```
[Link to Heading 1](#custom_id_for_first_heading)

[Link to Heading 2 in some document at the current site](/some/page/#custom_id_for_second_heading)

[Link to some heading with custom ID at an external site](https://some.site/path/to/the/page/#some_custom_id)
```

Epsconvert

EPSConvert is a tool to convert EPS images into PNG format.

Installation

```
$ pip install foliantcontrib.epsconvert
```

Config

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

```
preprocessors:
    - epsconvert
```

The preprocessor has a number of options:

```
preprocessors:
    - epsconvert:
        convert_path: convert
        cache_dir: !path .epsconvertcache
        image_width: 0
        targets:
            - pre
            - mkdocs
            - site
            - ghp
```

`convert_path` Path to `convert` binary. By default, it is assumed that you have this command in `PATH`. ImageMagick must be installed.

`cache_dir` Directory to store processed images. They may be reused later.

`image_width` Width of PNG images in pixels. By default (in case when the value is `0`), the width of each image is set by ImageMagick automatically. Default behavior is recommended. If the width is given explicitly, file size may increase.

`targets` Allowed targets for the preprocessor. If not specified (by default), the preprocessor applies to all targets.

EscapeCode and UnescapeCode

EscapeCode and UnescapeCode preprocessors work in pair.

EscapeCode finds in the source Markdown content the parts that should not be modified by any next preprocessors. Examples of content that should be left raw: fence code blocks, pre code blocks, inline code.

EscapeCode replaces these raw content parts with pseudo-XML tags recognized by UnescapeCode preprocessor.

EscapeCode saves raw content parts into files. Later, UnescapeCode restores this content from files.

Also, before the replacement, EscapeCode normalizes the source Markdown content to unify and simplify further operations. The preprocessor replaces `CRLF` with `LF`, removes excessive whitespace characters, provides trailing newline, etc.

Installation

To install EscapeCode and UnescapeCode preprocessors, run:

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

See more details below.

Integration with Foliant and Includes

You may call EscapeCode and UnescapeCode explicitly, but these preprocessors are integrated with Foliant core (since version 1.0.10) and with Includes preprocessor (since version 1.1.1).

The `escape_code` project's config option, if set to `true`, provides applying EscapeCode before all other preprocessors, and applying UnescapeCode after all other preprocessors. Also this option tells Includes preprocessor to apply EscapeCode to each included file.

In this mode EscapeCode and UnescapeCode preprocessors deprecate _unescape preprocessor.

```
>    **Note**
>
>    The preprocessor _unescape is a part of Foliant core. It allows to use pseudo-XML tags in code examples. If you want an opening tag not to be interpreted by any preprocessor, precede this tag with the `<` character. The preprocessor _unescape applies after all other preprocessors and removes such characters.
```

Config example:

```
title: My Awesome Project

chapters:
    - index.md
    ...

escape_code: true

preprocessors:
    ...
    - includes
    ...
...
```

If the `escape_code` option isn't used or set to `false`, backward compatibility mode is involved. In this mode EscapeCode and UnescapeCode aren't applied automatically, but _unescape preprocessor is applied.

The Python package that includes EscapeCode and UnescapeCode preprocessors is the dependence of Includes preprocessor since version 1.1.1. At the same time this package isn't a dependence of Foliant core. To use `escape_code` config option in Foliant core, you have to install the package with EscapeCode and UnescapeCode preprocessors separately.

Explicit Enabling

You may not to use the `escape_code` option and call the preprocessors explicitly:

```
preprocessors:
    - escapecode      # usually the first list item
    ...
    - unescapecode    # usually the last list item
```

Both preprocessors allow to override the path to the directory that is used to store temporary files:

```
preprocessors:
    - escapecode:
        cache_dir: !path .escapecodecache
    ...
    - unescapecode:
        cache_dir: !path .escapecodecache
```

The default values are shown in this example. EscapeCode and related UnescapeCode must work with the same cache directory.

Note that if you use Includes preprocessor, and the included content doesn't belong to the current Foliant project, there's no way to escape raw parts of this content before Includes preprocessor is applied.

Usage

Below you can see an example of Markdown content with code blocks and inline code.

````
# Heading

Text that contains some `inline code`.

Below is a fence code block, language is optional:

```python
import this
```

One more fence code block:

~~~
# This is a comment that should not be interpreted as a heading

print('Hello World')
~~~
````

```

And this is a pre code block:

    mov dx, hello;
    mov ah, 9;
    int 21h;
```

The preprocessor EscapeCode will do the following replacements:

```
# Heading

Text that contains some <escaped hash="2
bb20aeb00314e915ecfefd86d26f46a"></escaped>.

Below is a fence code block, language is optional:

<escaped hash="15e1e46a75ef29eb760f392bb2df4ebb"></escaped>

One more fence code block:

<escaped hash="91c3d3da865e24c33c4b366760c99579"></escaped>

And this is a pre code block:

<escaped hash="a1e51c9ad3da841d393533f1522ab17e"></escaped>
```

Escaped content parts will be saved into files located in the cache directory. The names of the files correspond the values of the `hash` attributes. For example, that's the content of the file `15e1e46a75ef29eb760f392bb2df4ebb.md`:

~~~
```python
import this
```
~~~

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.flags`:

```
preprocessors:
  - flags:
      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>...</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 a number of options with the following default values:

```
preprocessors:
    - flatten:
        flat_src_file_name: __all__.md
        keep_sources: false
```

`flat_src_file_name` Name of the flattened file that is created in the temporary working directory.

`keep_sources` Flag that tells the preprocessor to keep Markdown sources in the temporary working directory after flattening. If set to `false`, all Markdown files excepting the flattened will be deleted from the temporary working directory.

Note

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

Glossary

Glossary preprocessor collects terms and definitions from the files stated and inserts them to specified places of the document.

Installation

```
$ pip install foliantcontrib.glossary
```

Config

To enable the preprocessor, add `glossary` to `preprocessors` section in the project config.

```
preprocessors:
  - glossary
```

The preprocessor has a number of options (default values stated below):

```
preprocessors:
  - glossary:
        term_definitions: 'term_definitions.md'
        definition_mark: ':    '
        files_to_process: ''
```

`term_definitions` Local or remote file with terms and definitions in Pandoc definition_lists notation (by default this file stored in foliant project folder, but you can place it other folder). Also you can use includes in this file to join several glossary files. In this case `includes` preprocessor should be stated before `glossary` in `foliant.yml` preprocessors section. Note that if several definitions of one term are found, only first will be used.

`definition_mark` Preprocessor uses this string to determine, if the definition should be inserted here. `": "` for Pandoc definition_lists notation.

`files_to_process` You can set certain files to process by preprocessor.

Usage

Just add the preprocessor to the project config, set it up and enjoy the automatically collected glossary in your document.

Example

foliant.yml

```
...
chapters:
    - text.md

preprocessors:
...
    - includes
    - glossary
...
```

term_definitions.md

```
### Glossary

<include nohead="true">
    $https://git.repo/repo_name_1$src/glossary_1.md
</include>

<include nohead="true">
    $https://git.repo/repo_name_2$src/glossary_2.md
</include>
```

glossary_1.md from **repo_1**

```
### Glossary

Term 1

:   Definition 1

Term 2

:   Definition 2

Term 3

:   Definition 3
```

glossary_2.md from **repo_2**

```
### Glossary

Term 4

:   Definition 4

        { some code, part of Definition 4 }

    Third paragraph of definition 4.

Term 5

:   Definition 5
```

text.md

```
### Main chapter
```

```

Some text.

### Glossary

:   Term 1

:   Term 4

:   Term 2
```

__all__.md

```
### Main chapter

Some text.

### Glossary

Term 1

:   Definition 1


Term 4

:   Definition 4

        { some code, part of Definition 4 }

    Third paragraph of definition 4.


Term 2

:   Definition 2
```

Graphviz

pypi v1.1.1

Graphviz Diagrams Preprocessor for Foliant

Graphviz is an open source graph visualization tool. This preprocessor converts Graphviz diagram definitions in the source and converts them into images on the fly during project build.

Installation

```
$ pip install foliantcontrib.graphviz
```

Config

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

```
preprocessors:
    - graphviz
```

The preprocessor has a number of options:

```
preprocessors:
    - graphviz:
        cache_dir: !path .diagramscache
        graphviz_path: dot
        engine: dot
        format: png
        as_image: true
        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.

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

`graphviz_path` Path to Graphviz launcher. By default, it is assumed that you have the `dot` command in your `PATH`, but if Graphviz uses another command to launch, or if the `dot` launcher is installed in a custom place, you can define it here.

`engine` Layout engine used to process the diagram source. Available engines: (`circo`, `dot`, `fdp`, `neato`, `osage`, `patchwork`, `sfdp` `twopi`). Default: `dot`

`format` Output format of the diagram image. Available formats: tons of them. Default: `png`

`as_image` If `true` — inserts scheme into document as md-image. If `false` — inserts the file generated by GraphViz directly into the document (may be handy for `svg` images). Default: `true`

`params` Params passed to the image generation command:

```
preprocessors:
    - graphviz:
        params:
            Gdpi: 100
```

To see the full list of params, run the command that launches Graphviz, with `-?` command line option.

Usage

To insert a diagram definition in your Markdown source, enclose it between `<graphviz>...</graphviz>` tags:

```
’
Heres a diagram:

<graphviz>
```

```
    a -> b
</graphviz>
```

You can set any parameters in the tag options. Tag options have priority over the config options so you can override some values for specific diagrams while having the default ones set up in the config.

Tags also have two exclusive options: `caption` option — the markdown caption of the diagram image and `src` — path to diagram source (relative to current file).

> If `src` tag option is supplied, tag body is ignored. Diagram source is loaded from external file.

```
Diagram with a caption:

<graphviz caption="Deployment diagram"
          params="Earrowsize: 0.5"
          src="diags/sample.gv">
</graphviz>
```

> Note that command params listed in the `params` option are stated in YAML format. Remember that YAML is sensitive to indentation so for several params it is more suitable to use JSON-like mappings: `{key1: 1, key2: 'value2'}`.

History

History is a preprocessor that generates single linear history of releases for multiple Git repositories based on their changelog files, tags, or commits. The history may be represented as Markdown, and as RSS feed.

Installation

```
$ pip install foliantcontrib.history
```

Config

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

```
preprocessors:
    - history
```

The preprocessor has a number of options with the following default values:

```
- history:
    repos: []
    revision: master
    name_from_readme: false
    readme: README.md
    from: changelog
    merge_commits: true
    changelog: changelog.md
    source_heading_level: 1
    target_heading_level: 1
    target_heading_template: '[%date%] [%repo%](%link%) %version%'
    date_format: year_first
    limit: 0
    rss: false
    rss_file: rss.xml
    rss_title: 'History of Releases'
    rss_link: ''
    rss_description: ''
    rss_language: en-US
    rss_item_title_template: '%repo% %version%'
```

`repos` List of URLs of Git repositories that it's necessary to generate history for.

Example:

```
repos:
    - https://github.com/foliant-docs/foliant.git
    - https://github.com/foliant-docs/foliantcontrib.
includes.git
```

`revision` Revision or branch name to use. Branches that are used for stable releases must have the same names in all listed repositories.

`name_from_readme` Flag that tells the preprocessor to try to use the content of the first heading of README file in each listed repository as the repo name. If the flag set to `false`, or an attempt to get the first heading content is unsuccessful, the repo name will be based on the repo URL.

`readme` Path to README file. README files must be located at the same paths in all listed repositories.

`from` Data source to generate history: `changelog`—changelog file, `tags`—tags, `commits`—all commits. Data sources of the same type will be used for all listed repositories.

`merge_commits` Flag that tells the preprocessor to include merge commits into history when `from: commits` is used.

`changelog` Path to changelog file. Changelogs must be located at the same paths in all listed repositories.

`source_heading_level` Level of headings that precede descriptions of releases in the source Markdown content. It must be the same for all listed repositories.

`target_heading_level` Level of headings that precede descriptions of releases in the target Markdown content of generated history.

`target_heading_template` Template for top-level headings in the target Markdown content. You may use any characters, and the variables: `%date%`—date, `%repo%`—repo name, `%link%`—repo URL, `%version%`—version data (content of source changelog heading, tag value, or commit hash).

`date_format` Output date format to use in the target Markdown content. If the default value `year_first` is used, the date "September 4, 2019" will be represented as `2019-09-04`. If the `day_first` value is used, this date will be represented as `04.09.2019`.

`limit` Maximum number of items to include into the target Markdown content; `0` means no limit.

`rss` Flag that tells the preprocessor to export the history into RSS feed. Note that the parameters `target_heading_level`, `target_heading_template`, `date_format`, and `limit` are applied to Markdown content only, not to RSS feed content.

`rss_file` Subpath to the file with RSS feed. It's relative to the temporary working directory during building, to the directory of built project after building, and to the `rss_link` value in URLs.

`rss_title` RSS channel title.

`rss_link` RSS channel link, e.g. `https://foliant-docs.github.io/docs/`. If the `rss` parameter value is `rss.xml`, the RSS feed URL will be `https://foliant-docs.github.io/docs/rss.xml`.

`rss_description` RSS channel description.

`rss_language` RSS channel language.

`rss_item_title_template` Template for titles of RSS feed items. You may use any characters, and the variables: `%repo%`—repo name, `%version%`—version data.

Usage

To insert some history into Markdown content, use the `<history></history>` tags:

```
Some optional content here.

<history></history>

More optional content.
```

If no attributes specified, the values of options from the project config will be used.

You may override each config option value with the attribute of the same name. Example:

```
<history
    repos="https://github.com/foliant-docs/foliantcontrib.
mkdocs.git"
    revision="develop"
    limit="5"
    rss="true"
    rss_file="some_another.xml"
    ...
>
</history>
```

ImageMagick

This tool provides additional processing of images that referred in Markdown source, with ImageMagick.

Installation

```
$ pip install foliantcontrib.imagemagick
```

Config

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

```
preprocessors:
    - imagemagick
```

The preprocessor has a number of options with the following default values:

```
preprocessors:
    - imagemagick:
        convert_path: convert
        cache_dir: .imagemagickcache
```

`convert_path` Path to `convert` binary, a part of ImageMagick.

`cache_dir` Directory to store processed images. These files can be reused later.

Usage

Suppose you want to apply the following command to your picture `image.eps`:

```
$ convert image.eps -resize 600 -background Orange label:'
Picture' +swap -gravity Center -append image.jpg
```

This command takes the source EPS image `image.eps`, resizes it, puts a text label over the picture, and writes the result into new file `image.jpg`. The suffix of output file name specifies that the image must be converted into JPEG format.

To use the ImageMagick preprocessor to do the same, enclose one or more image references in your Markdown source between `<magick>` and `</magick>` tags.

```
<magick command_params="-resize 600 -background Orange label
:'Picture' +swap -gravity Center -append" output_format="jpg
">
(leading exclamation mark here)[Optional Caption](image.eps)
</magick>
```

Use `output_format` attribute to specify the suffix of output file name. The whole output file name will be generated automatically.

Use `command_params` attribute to specify the string of parameters that should be passed to ImageMagick `convert` binary.

Instead of using `command_params` attribute, you may specify each parameter as its own attribute with the same name:

```
<magick resize="600" background="Orange label:'Picture' +
swap" gravity="Center" append="true" output_format="jpg">
(leading exclamation mark here)[Optional Caption](image.eps)
</magick>
```

ImgCaptions

ImgCaptions is a preprocessor that generates visible captions for the images from alternative text descriptions of the images. The preprocessor is useful in projects built with MkDocs or another backend that provides HTML output.

Installation

```
$ pip install foliantcontrib.imgcaptions
```

Usage

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

```
preprocessors:
    - imgcaptions
```

The preprocessor supports the following options:

```
    - imgcaptions:
        stylesheet_path: !path imgcaptions.css
        targets:
            - pre
            - mkdocs
            - site
            - ghp
```

`stylesheet_path` Path to the CSS stylesheet file. This stylesheet should define rules for the `.image_caption` class. Default path is `imgcaptions.css`. If stylesheet file does not exist, default built-in stylesheet will be used.

`targets` Allowed targets for the preprocessor. If not specified (by default), the preprocessor applies to all targets.

Image definition example:

```
(leading exclamation mark here)[My Picture](picture.png)
```

This Markdown source will be finally transformed into the HTML code:

```
<p><img alt="My Picture" src="picture.png"></p>
<p class="image_caption">My Picture</p>
```

(Note that ImgCaptions preprocessor does not convert Markdown syntax into HTML; it only inserts HTML tags `<p class="image_caption">My Picture</p>` into Markdown code after the image definitions. Empty alternative text descriptions are ignored.)

ImgConvert

ImgConvert is a tool to convert images from an arbitrary format into PNG.

Installation

```
$ pip install foliantcontrib.imgconvert
```

Config

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

```
preprocessors:
    - imgconvert
```

The preprocessor has a number of options with the following default values:

```
preprocessors:
    - imgconvert:
        convert_path: convert
        cache_dir: !path .imgconvertcache
        image_width: 0
        formats: {}
```

`convert_path` Path to `convert` binary. By default, it is assumed that you have this command in `PATH`. ImageMagick must be installed.

`cache_dir` Directory to store processed images. They may be reused later.

`image_width` Width of PNG images in pixels. By default (in case when the value is `0`), the width of each image is set by ImageMagick automatically. Default behavior is recommended. If the width is given explicitly, file size may increase.

`formats` Settings that apply to each format of source images.

The `formats` option may be used to define lists of targets for each format. If targets for a format are not specified explicitly, the preprocessor will be applied to all targets.

Example:

```
formats:
    eps:
        targets:
            - site
    svg:
        targets:
            - docx
```

Formats should be named in lowercase.

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 Git 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.

> **Note**
> Aliases are available only within the legacy syntax of include statements (see below).

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

```
- includes:
    aliases:
        foo: https://github.com/boo/bar.git
        baz: https://github.com/foo/far.git#develop
```

you can include the content of doc.md files from these repositories using the following syntax:

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

<include>$baz#master$path/to/doc.md</include>
```

Note that in the second example the default revision (develop) will be overridden with the custom one (master).

Usage

The preprocessor allows two syntax variants for include statements.

The **legacy** syntax is simpler and shorter but less flexible. There are no plans to extend it.

The **new** syntax introduced in version 1.1.0 is stricter and more flexible. It is more suitable for complex cases, and it can be easily extended in the future. This is the preferred syntax.

Both variants of syntax use the <include>...</include> tags.

If the included file is specified between the tags, it's the legacy syntax. If the file is referenced in the tag attributes (src, repo_url, path), it's the new one.

The New Syntax

To enforce using the new syntax rules, put no content between <include>...</include> tags, and specify a local file or a file in a remote Git repository in tag attributes.

To include a local file, use the src attribute:

```
Text below is taken from another document.

<include src="path/to/another/document.md"></include>
```

To include a file from a remote Git repository, use the repo_url and path attributes:

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

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

You have to specify the full remote repository URL in the `repo_url` attribute, aliases are not supported here.

Optional branch or revision can be specified in the `revision` attribute:

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

<include repo_url="https://github.com/foo/bar.git" revision
="develop" path="path/to/doc.md"></include>
```

Attributes

`src` Path to the local file to include.

`repo_url` Full remote Git repository URL without a revision.

`path` Path to the file inside the remote Git repository.

> **Note**
> If you are using the new syntax, the `src` attribute is required to include a local file, and the `repo_url` and `path` attributes are required to include a file from a remote Git repository. All other attributes are optional.

> **Note**
> Foliant 1.0.9 supports the processing of attribute values as YAML. You can precede the values of attributes by the `!path`, `!project_path`, and `!rel_path` modifiers (i.e. YAML tags). These modifiers can be useful in the `src`, `path`, and `project_root` attributes.

`revision` Revision of the Git repository.

`from_heading` Full content of the starting heading when it's necessary to include some part of the referenced file content.

`to_heading` Full content of the ending heading when it's necessary to include some part of the referenced file content.

`from_id` ID of the starting heading or starting anchor when it's necessary to include some part of the referenced file content. The `from_id` attribute has higher priority than `from_heading`.

`to_id` ID of the ending heading or ending anchor when it's necessary to include some part of the referenced file content. The `to_id` attribute has higher priority than `to_heading`.

`to_end` Flag that tells the preprocessor to cut to the end of the included content. Otherwise, if `from_heading` or `from_id` is specified, the preprocessor cuts the included content to the next heading of the same level as the starting heading, or the heading that precedes the starting anchor.

Example:

```
## Some Heading {#custom_id}

<anchor>one_more_custom_id</anchor>
```

Here `Some Heading {#custom_id}` is the full content of the heading, `custom_id` is its ID, and `one_more_custom_id` is the ID of the anchor.

Optional Attributes Supported in Both Syntax Variants

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

```
# Title

## Subtitle

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

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

```
# My Custom Heading

<include src="other.md" from_heading="Original Heading"
 nohead="true"></include>
```

Default is `false`.

By default, the starting heading is included to the output, and the ending heading is not. Starting and ending anchors are never included into the output.

`inline` Flag that tells the preprocessor to replace sequences of whitespace characters of many kinds (including `\r`, `\n`, and `\t`) with single spaces ( ) in the included content, and then to strip leading and trailing spaces. It may be useful in single-line table cells. Default value is `false`.

`project_root` Path to the top-level ("root") directory of Foliant project that the included file belongs to. This option may be needed to resolve the `!path` and `!project_path` modifiers in the included content properly.

> **Note**
>
> By default, if a local file is included, `project_root` points to the top-level directory of the current Foliant project, and if a file in a remote Git repository is referenced, `project_root` points to the top-level directory of this repository. In most cases you don't need to override the default behavior.

Different 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 src="other.md" from_heading="Original Heading"
sethead="1" nohead="true"></include>
```

The Legacy Syntax

This syntax was the only supported in the preprocessor up to version 1.0.11. It's weird and cryptic, you had to memorize strange rules about `$`, `#` and stuff. The new syntax described above is much cleaner.

The legacy syntax is kept for backward compatibility. To use it, put the reference to the included file between `<include>...</include>` tags.

Local path example:

```
Text below is taken from another document.

<include>path/to/another/document.md</include>
```

The path may be either relative to currently processed Markdown file or absolute.

To include a document from a remote Git repository, put its URL between `$`s before 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>
```

In the legacy syntax, problems may occur with the use of `$`, `#`, and `:` characters in filenames and headings, since these characters may be interpreted as delimeters.

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:

```yaml
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:

```yaml
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](<if backends="pandoc">#another_page</if><if backends="mkdocs">another_page.md</if>).
```

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><if backends="mkdocs">{1}</if>
```

And use it in the source:

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

Mermaid

pypi v1.0.0

Mermaid Diagrams Preprocessor for Foliant

Mermaid is an open source diagram visualization tool. This preprocessor converts Mermaid diagram definitions in your Markdown files into images on the fly during project build.

Installation

```
$ pip install foliantcontrib.mermaid
```

Please note that to use this preprocessor you will also need to install Mermaid and Mermaid CLI:

```
$ npm install mermaid # installs locally
$ npm install mermaid.cli
```

Config

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

```
preprocessors:
    - mermaid
```

The preprocessor has a number of options:

```
preprocessors:
    - mermaid:
        cache_dir: !path .diagramscache
        mermaid_path: !path node_modules/.bin/mmdc
        format: svg
        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.

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

`mermaid_path` Path to Mermaid CLI binary. If you installed Mermaid locally this parameter is required. Default: `mmdc`.

`format` Generated image format. Available: `svg`, `png`, `pdf`. Default `svg`.

`params` Params passed to the image generation command:

```
preprocessors:
    - mermaid:
        params:
            theme: forest
```

> To see the full list of available params, run `mmdc -h` or check here.

Usage

To insert a diagram definition in your Markdown source, enclose it between `<mermaid>...</mermaid>` tags:

```
'
Heres a diagram:

<mermaid>
graph TD;
    A-->B;
</mermaid>
```

You can set any parameters in the tag options. Tag options have priority over the config options so you can override some values for specific diagrams while having the default ones set up in the config.

Tags also have an exclusive option `caption` — the markdown caption of the diagram image.

```
Diagram with a caption:

<mermaid caption="Deployment diagram"
          params="theme: dark">
</mermaid>
```

> Note that command params listed in the `params` option are stated in YAML format. Remember that YAML is sensitive to indentation so for several params it is more suitable to use JSON-like mappings: `{key1: 1, key2: 'value2'}`.

MultilineTables

This preprocessor converts tables to multiline and grid format before creating document (very useful especially for pandoc processing). It helps to make tables in doc and pdf formats more proportional — column with more text in it will be more wide. Also it helps whith processing of extremely wide tables with pandoc. Convertation to the grid format allows arbitrary cell' content (multiple paragraphs, code blocks, lists, etc.).

Installation

```
$ pip install foliantcontrib.multilinetables
```

Config

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

```
preprocessors:
  - multilinetables
```

The preprocessor has a number of options (best values set by default):

```
preprocessors:
    - multilinetables:
        rewrite_src_files: false
        min_table_width: 100
        keep_narrow_tables: true
        table_columns_to_scale: 3
        enable_hyphenation: false
        hyph_combination: '<br>'
        convert_to_grid: false
        targets:
            - docx
            - pdf
```

`rewrite_src_file` You can update source files after each use of preprocessor. Be careful, previous data will be deleted.

`min_table_width` Wide markdown tables will be shrinked to this width in symbols. This parameter affects scaling - change it if table columns are merging.

`keep_narrow_tables` If `true` narrow tables will not be stretched to minimum table width.

`table_columns_to_scale` Minimum amount of columns to process the table.

`enable_hyphenation` Switch breaking text in table cells with the tag set in `hyph_combination`. Good for lists, paragraphs, etc.

`hyph_combination` Custom tag to break a text in multiline tables.

`convert_to_grid` If `true` tables will be converted to the grid format, that allows arbitrary cell' content (multiple paragraphs, code blocks, lists, etc.).

`targets` Allowed targets for the preprocessor. If not specified (by default), the preprocessor applies to all targets.

Usage

Just add preprocessor to the project config and enjoy the result.

Pgsqldoc

This preprocessor generates simple documentation of a PostgreSQL database based on its structure. It uses Jinja2 templating engine for customizing the layout and PlantUML for drawing the database scheme.

Installation

```
$ pip install foliantcontrib.pgsqldoc
```

Config

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

```
preprocessors:
    - pgsqldoc
```

The preprocessor has a number of options:

```
preprocessors:
    - pgsqldoc:
        host: localhost
        port: 5432
        dbname: postgres
        user: postgres
        password: ''
        draw: false
        filters:
            ...
        doc_template: pgsqldoc.j2
        scheme_template: scheme.j2
```

`host` PostgreSQL database host address. Default: `localhost`

`port` PostgreSQL database port. Default: `5432`

`dbname` PostgreSQL database name. Default: `postgres`

`user` PostgreSQL user name. Default: `postgres`

`passwrod` PostgreSQL user password.

`draw` If this parameter is `true` — preprocessor would generate scheme of the database and add it to the end of the document. Default: `false`

`filters` SQL-like operators for filtering the results. More info in the **Filters** section.

`doc_template` Path to jinja-template for documentation. Path is relative to the project directory. Default: `pgsqldoc.j2`

`scheme_template` Path to jinja-template for scheme. Path is relative to the project directory. Default: `scheme.j2`

Usage

Add a `<pgsqldoc></pgsqldoc>` tag at the position in the document where the generated documentation of a PostgreSQL database should be inserted:

```
### Introduction

This document contains the most awesome automatically
generated documentation of our marvellous database.
```

```

<pgsqldoc></pgsqldoc>
```

Each time the preprocessor encounters the tag `<pgsqldoc></pgsqldoc>` it inserts the whole generated documentation text instead of it. The connection parameters are taken from the config-file.

You can also specify some parameters (or all of them) in the tag options:

```
### Introduction

Introduction text for database documentation.

<pgsqldoc draw="true"
          host="11.51.126.8"
          port="5432"
          dbname="mydb"
          user="scott"
          password="tiger">
</pgsqldoc>
```

Tag parameters have the highest priority.

This way you can have documentation for several different databases in one foliant project (even in one md-file if you like it so).

Filters

You can add filters to exclude some tables from the documentation. Pgsqldocs supports several SQL-like filtering operators and a determined list of filtering fields.

You can switch on filters either in foliant.yml file like this:

```
preprocessors:
  - pgsqldoc:
    filters:
      eq:
        schema: public
      regex:
```

```
      table_name: 'main_.+'
```

or in tag options using the same yaml-syntax:

```

<pgsqldoc filters="
eq:
    schema: public
  regex:
    table_name: 'main_.+'">
</pgsqldoc>
```

List of currently supported operators:

operator	SQL equivalent	description	value
`eq`	`=`	equals	literal
`not_eq`	`!=`	does not equal	literal
`in`	`IN`	contains	list
`not_in`	`NOT IN`	does not contain	list
`regex`	`~`	matches regular expression	literal
`not_regex`	`!~`	does not match regular expression	literal

List of currently supported filtering fields:

field	description
schema	filter by PostgreSQL database schema
table_name	filter by database table names

The syntax for using filters in configuration files is following:

```
filters:
  <operator>:
    <field>: value
```

If `value` should be list like for `in` operator, use YAML-lists instead:

```
filters:
  in:
    schema:
      - public
      - corp
```

About Templates

The structure of generated documentation is defined by jinja-templates. You can choose what elements will appear in the documentation, change their positions, add constant text, change layouts and more. Check the Jinja documentation for info on all cool things you can do with templates.

If you don't specify path to templates in the config-file and tag-options pgsqldoc will use default paths:

- `<Project_path>/pgsqldoc.j2` for documentation template;
- `<Project_path>/scheme.j2` for database scheme source template.

If pgsqldoc can't find these templates in the project dir it will generate default templates and put them there.

So if you accidentally mess things up while experimenting with templates you can always delete your templates and run preprocessor — the default ones will appear in the project dir. (But only if the templates are not specified in config-file or their names are the same as defaults).

One more useful thing about default templates is that you can find a detailed description of the source data they get from pgsqldoc in the beginning of the template.

Plantuml

PlantUML is a tool to generate diagrams from plain text. This preprocessor finds PlantUML diagrams definitions in the source and converts them into images on the fly during project build.

Installation

```
$ pip install foliantcontrib.plantuml
```

Config

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

```
preprocessors:
    - plantuml
```

The preprocessor has a number of options:

```
preprocessors:
    - plantuml:
        cache_dir: !path .diagramscache
        plantuml_path: plantuml
        params:
            ...
        parse_raw: true
```

`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.

`plantuml_path` Path to PlantUML launcher. By default, it is assumed that you have the command `plantuml` in your PATH, but if PlantUML uses another command to launch, or if the `plantuml` launcher is installed in a custom place, you can define it here.

`params` Params passed to the image generation command:

```
preprocessors:
    - plantuml:
        params:
            config: !path plantuml.cfg
```

To see the full list of params, run the command that launches PlantUML, with `-h` command line option.

`parse_raw` If this flag is `true`, the preprocessor will also process all PlantUML diagrams which are not wrapped in `<plantuml>...</plantuml>` tags. Default value is `false`.

Usage

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

```
'
Heres a diagram:

<plantuml>
    @startuml
        ...
    @enduml
</plantuml>
```

To set a caption, use `caption` option:

```
Diagram with a caption:

<plantuml caption="Sample diagram from the official site">
    @startuml
        ...
    @enduml
</plantuml>
```

You can override `params` values from the preprocessor config for each diagram. Also you can use `format` alias for `-t*` params:

```
By default, diagrams are in PNG. But this diagram is in EPS:

<plantuml caption="Vector diagram" format="eps">
```

```
    @startuml
        ...
    @enduml
</plantuml>
```

Sometimes it can be necessary to process auto-generated documents that contain multiple PlantUML diagrams definitions without using Foliant-specific tags syntax. Use the `parse_raw` option in these cases.

Replace

Replace preprocessor reads the dictionary (yaml format) placed in foliant project folder and changes one word to another in created document.

Installation

```
$ pip install foliantcontrib.replace
```

Config

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

```
preprocessors:
  - replace
```

The preprocessor has two options (default values stated):

```
preprocessors:
  - replace:
      dictionary_filename: replace_dictionary.yml
      with_confirmation: false
```

`dictionary_filename` File in foliant project folder with dictionary in it (*replace_-dictionary.yml* by default).

`with_confirmation` if `true` you will be prompted to confirm any changes.

Dictionary format

Dictionary stores data in yaml format. It has two sections — with words and with regular expressions. You can pass the lambda function in `regexs` section. For example:

```
words:
  cod: CoD
  epg: EPG
  vod: VoD
regexs:
  '!\w*!': ''
  '\. *(\w)': 'lambda x: x.group(0).upper()'
```

Usage

Just add the preprocessor to the project config, set the dictionary and enjoy the result.

RepoLink

This preprocessor allows to add into each Markdown source a hyperlink to the related file in Git repository. The hyperlink appears after the first heading of the document.

The preprocessor emulates MkDocs behavior and supports the same options `repo_url` and `edit_uri` as MkDocs. Applying of the preprocessor to subprojects allows to get links to separate repositories from different pages of a single site (or a single MkDocs project).

Installation

RepoLink preprocessor is a part of MultiProject extension:

```
$ pip install foliantcontrib.multiproject
```

Usage

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

```
preprocessors:
    - repolink
```

The preprocessor has a number of options:

```
preprocessors:
    - repolink:
        repo_url: https://github.com/foliant-docs/docs/
        edit_uri: /blob/master/src/
        link_text: "&#xE3C9;"
        link_title: View the source file
        link_html_attributes: "class=\"md-icon md-content__icon\" style=\"margin: -7.5rem 0\""
        targets:
            - pre
```

`repo_url` URL of the related repository. Default value is an empty string; in this case the preprocessor does not apply. Trailing slashes do not affect.

`edit_uri` Revision-dependent part of URL of each file in the repository. Default value is `/blob/master/src/`. Leading and trailing slashes do not affect.

`link_text` Hyperlink text. Default value is `Edit this page`.

`link_title` Hyperlink title (the value of `title` HTML attribute). Default value is also `Edit this page`.

`link_html_attributes` Additional HTML attributes for the hyperlink. By using CSS in combination with `class` attribute, and/or `style` attribute, you may customize the presentation of your hyperlinks. Default value is an empty string.

`targets` Allowed targets for the preprocessor. If not specified (by default), the preprocessor applies to all targets.

You may override the value of the `edit_uri` config option with the `FOLIANT_REPOLINK_EDIT_URI` system environment variable. It can be useful in some non-stable testing or staging environments.

RunCommands

RunCommands is a preprocessor that allows to execute a sequence of arbitrary external commands.

Installation

```
$ pip install foliantcontrib.runcommands
```

Usage

To enable the preprocessor, add `runcommands` to `preprocessors` section in the project config, and specify the commands to run:

```
preprocessors:
    - runcommands:
        commands:
            - ./build.sh
            - echo "Hello World" > ${WORKING_DIR}/hello.txt
        targets:
            - pre
            - tex
            - pdf
            - docx
```

`commands` Sequence of system commands to execute one after the other.

`targets` Allowed targets for the preprocessor. If not specified (by default), the preprocessor applies to all targets.

Supported environment variables

You may use the following environment variables in your commands:

- `${PROJECT_DIR}` — full path to the project directory, e.g. `/usr/src/app`;
- `${SRC_DIR}` — full path to the directory that contains Markdown sources, e.g. `/usr/src/app/src`;
- `${WORKING_DIR}` — full path to the temporary directory that is used by preprocessors, e.g. `/usr/src/app/__folianttmp__`;

- `${BACKEND}` — currently used backend, e.g. `pre`, `pandoc`, or `mkdocs`;
- `${TARGET}` — current target, e.g. `site`, or `pdf`.

SwaggerDoc

pypi v1.2.0

Swagger API Docs Generator for Foliant

This preprocessor generates Markdown documentation from Swagger spec files . It uses Jinja2 templating engine or widdershins for generating Markdown from swagger spec files.

Installation

```
$ pip install foliantcontrib.swaggerdoc
```

Config

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

```
preprocessors:
    - swaggerdoc
```

The preprocessor has a number of options:

```
preprocessors:
    - swaggerdoc:
        json_url: http://localhost/swagger.json
        json_path: swagger.json
        additional_json_path: tags.json
        mode: 'jinja'
        template: swagger.j2
        environment: env.yaml
```

`json_url` URL to Swagger spec file. If it is a list — preprocessor takes the first url which works.

even though the parameters are called *json*_url and *json*_path, yaml format is supported too. Parameters may be softly renamed in future.

`json_path` Local path to Swagger spec file (relative to project dir).

If both url and path are specified — preprocessor first tries to fetch JSON from url, and then (if that fails) looks for the file on local path.

`additional_json_path` Only for `jinja` mode. Local path to swagger spec file with additional info (relative to project dir). It will be merged into original spec file, *not overriding existing fields.*

`mode` Determines how the Swagger spec file would be converted to markdown. Should be one of: `jinja`, `widdershins`. Default: `widdershins`

`jinja` mode is deprecated. It may be removed in future

`template` Only for `jinja` mode. Path to jinja-template for rendering the generated documentation. Path is relative to the project directory. If no template is specified preprocessor will use default template (and put it into project dir if it was missing). Default: `swagger.j2`

`environment` Only for `widdershins` mode. Parameters for widdershins converter. You can either pass a string containing relative path to YAML or JSON file with all parameters (like in example above) or specify all parameters in YAML format under this key. More info on widdershins parameters.

Usage

Add a `<swaggerdoc></swaggerdoc>` tag at the position in the document where the generated documentation should be inserted:

```
### Introduction

This document contains the automatically generated
documentation of our API.
```

```

<swaggerdoc></swaggerdoc>
```

Each time the preprocessor encounters the tag `<swaggerdoc></swaggerdoc>` it inserts the whole generated documentation text instead of it. The path or url to Swagger spec file are taken from foliant.yml.

You can also specify some parameters (or all of them) in the tag options:

```
### Introduction

Introduction text for API documentation.

<swaggerdoc json_url="http://localhost/swagger.json"
            mode="jinja"
            template="swagger.j2">
</swaggerdoc>

<swaggerdoc json_url="http://localhost/swagger.json"
            mode="widdershins"
            environment="env.yml">
</swaggerdoc>
```

Tag parameters have the highest priority.

This way you can have documentation from several different Swagger spec files in one foliant project (even in one md-file if you like it so).

Customizing output

Jinja

> `jinja` mode is deprecated. It may be removed in future

In `jinja` mode the output markdown is generated by the Jinja2 template. In this template all fields from Swagger spec file are available under the dictionary named `swagger_data`.

To customize the output create a template which suits your needs. Then supply the path to it in the `template` parameter.

If you wish to use the default template as a starting point, build the foliant project with `swaggerdoc` preprocessor turned on. After the first build the default template will appear in your foliant project dir under name `swagger.j2`.

Widdershins

In `widdershins` mode the output markdown is generated by widdershins Node.js application. It supports customizing the output with doT.js templates.

1. Clone the original widdershins repository and modify the templates located in one of the subfolders in the **templates** folder.
2. Save the modified templates somewhere near your foliant project.
3. Specify the path to modified templates in the `user_templates` field of the `environment` configuration. For example, like this:

```
preprocessors:
    - swaggerdoc:
        json_path: swagger.yml
        environment:
            user_templates: !path ./widdershins_templates/
```

TemplateParser

pypi v1.0.2

TemplateParser preprocessor for Foliant

Preprocessor which allows to use templates in Foliant source files. Preprocessor now supports only Jinja2 templating engine, but more can be added easily.

Installation

```
$ pip install foliantcontrib.templateparser
```

Config

All params that are stated in foliant.yml are considered global params. All of them may be overriden in template tag options, which have higher priority.

```yaml
preprocessors:
    - templateparser:
        engine: jinja2
        engine_params:
            root: '/usr/src/app'
        context:
            param1: 1008
            param2: 'Kittens'
        ext_context: context.yml
        param3: 'Puppies'
```

`engine` name of the template engine which will be used to process template. Supported engines right now: `jinja2`.

`engine_params` dictionary with parameters which will be transfered to the template engine.

`context` dictionary with variables which will be redirected to the template.

`ext_context` path to YAML- or JSON-file with context dictionary. (relative to current md-file)

All parameters with other names are also transfered to the template, as if they appeared inside the `context` *dictionary. (* `param3` *in the above example)*

> Please note that even if this may seem convenient, it is preferred to include template variables in the `context` dictionary, as in future more reserved parameters may be added which may conflict with your stray variables.

If some variable names overlap among these methods to supply context, preprocessor uses this priority order:

1. Context dictionary.
2. Stray variables.
3. External context file.

Usage

To use the template in a Markdown file just insert a tag of the template engine name, for example:

```
This is ordinary markdown text.
<jinja2>
This is a Jinja2 template:
I can count to five!
{% for i in range(5) %}{{ i + 1 }}{% endfor %}
</jinja2>
```

After making a document with Foliant this will be transformed to:

```
This is ordinary markdown text.

This is a Jinja2 template:
I can count to five!
12345
```

You can also use a general `<template>` tag, but in this case you have to specify the engine you want to use in the `engine` parameter:

```
This is ordinary markdown text.
<template engine="jinja2">
This is a Jinja2 template:
I can count to five!
{% for i in range(5) %}{{ i + 1 }}{% endfor %}
</template>
```

Sending variables to template

To send a variable to template, add them into the `context` option. This option accepts `yaml` dictionary format.

Please note that foliant doesn't support multiline tag options yet, so use one-line dictionary format {"key1": value1, ...}

```
<jinja2 context="{'name': Andy, 'age': 8}">
Hi, my name is {{name}}!
I am {{age}} years old.
{% for prev in range(age - 1, 0, -1) %}
The year before I was {{prev}} years old.
{% endfor %}
</jinja2>
```

Result:

```
Hi, my name is Andy!
I am 8 years old.

The year before I was 7 years old.

The year before I was 6 years old.

The year before I was 5 years old.

The year before I was 4 years old.

The year before I was 3 years old.

The year before I was 2 years old.

The year before I was 1 years old.
```

Extends and includes

Extends and includes work in templates. The path of the extending\included file is relative to the Markdown file where the template lives.

In Jinja2 engine you can override the path of the included\extended files with `root` engine_param. **Note that this param is relative to project root.**

Testrail

TestRail preprocessor collects test cases from TestRail project and adds to your testing procedure document.

Installation

```
$ pip install foliantcontrib.testrail
```

Config

To enable the preprocessor, add `testrail` to `preprocessors` section in the project config. The preprocessor has a number of options (best values are set by default where possible):

```
preprocessors:
  - testrail:
    testrail_url: http://testrails.url
                                \\ Required
    testrail_login: username
                                \\ Required
    testrail_pass: password
                                \\ Required
    project_id: 35
                                \\ Required
    suite_ids:
                                \\ Optional
    section_ids:
                                \\ Optional
    exclude_suite_ids:
                                \\ Optional
    exclude_section_ids:
                                \\ Optional
    exclude_case_ids:
                                \\ Optional
    filename: test_cases.md
                                \\ Optional
    rewrite_src_file: false
                                \\ Optional
    template_folder: case_templates
                                \\ Optional
    section_header: Testing program
                                \\ Recommended
    add_std_table: true
                                \\ Optional
    std_table_header: Table with testing results
                                \\ Recommended
    std_table_column_headers: №; Priority; Platform; ID;
Test case name; Result; Comment    \\ Recommended
    add_suite_headers: true
                                \\ Optional
    add_section_headers: true
                                \\ Optional
    add_case_id_to_case_header: false
                                \\ Optional
    add_case_id_to_std_table: false
                                \\ Optional
    multi_param_name:
                                \\ Optional
    multi_param_select:
                                \\ Optional
    multi_param_select_type: any
                                \\ Optional
    add_cases_without_multi_param: true
                                \\ Optional
    add_multi_param_to_case_header: false
                                \\ Optional
    add_multi_param_to_std_table: false
                                \\ Optional
    checkbox_param_name:
                                \\ Optional
    checkbox_param_select_type: checked
                                \\ Optional
    choose_priorities:
                                \\ Optional
    add_priority_to_case_header: false
                                \\ Optional
    add_priority_to_std_table: false
                                \\ Optional
    resolve_urls: true
                                \\ Optional
    screenshots_url: https://gitlab_repository.url
                                \\ Optional
    screenshots_ext: .png
                                \\ Optional
    print_case_structure: true
                                \\ For debugging
```

`testrail_url` URL of TestRail deployed.

`testrail_login` Your TestRail username.

`testrail_pass` Your TestRail password.

`project_id` TestRail project ID. You can find it in the project URL, for example http://testrails.url/index.php?/projects/overview/17 <-.

`suite_ids` If you have several suites in your project, you can download test cases from certain suites. You can find suite ID in the URL again, for example http://testrails.url/index.php?/suites/view/63... <-.

`section_ids` Also you can download any sections you want regardless of it's level. Just keep in mind that this setting overrides previous *suite_ids* (but if you set *suite_ids* and then *section_ids* from another suite, nothing will be downloaded). And suddenly you can find section ID in it's URL, for example http://testrails.url/index.php?/suites/view/124&group_by=cases:section_id&group_order=asc&group_id=3926 <-.

`exclude_suite_ids` You can exclude any suites (even stated in *suite_ids*) from the document.

`exclude_section_ids` The same with the sections.

`exclude_case_ids` And the same with the cases.

`filename` Path to the test cases file. It should be added to project chapters in *foliant.yml*. Default: *test_cases.md*. For example:

```
title: &title Test procedure

chapters:
    - intro.md
    - conditions.md
    - awesome_test_cases.md <- This one for test cases
    - appendum.md

preprocessors:
  - testrail:
    testrail_url: http://testrails.url
    testrail_login: username
    testrail_pass: password
    project_id: 35
    filename: awesome_test_cases.md
```

`rewrite_src_file` You can update (*true*) test cases file after each use of preprocessor. Be careful, previous data will be deleted.

`template_folder` Preprocessor uses Jinja2 templates to compose the file with test cases. Here you can find documentation: http://jinja.pocoo.org/docs/2.10/ . You can store templates in folder inside the foliant project, but if it's not default *case_templates* you have to write it here.

If this parameter not set and there is no default *case_templates* folder in the project, it will be created automatically with two jinja files for TestRail templates by default – *Test Case (Text)* with *template_id=1* and *Test Case (Steps)* with *template_id=2*.

You can create TestRail templates by yourself in *Administration* panel, *Customizations* section, *Templates* part. Then you have to create jinja templates whith the names *{template_id}.j2* for them. For example, file *2.j2* for *Test Case (Steps)* TestRail template:

```

```

```
{% if case['custom_steps_separated'][0]['content'] %}
{% if case['custom_preconds'] %}
**Preconditions:**

{{ case['custom_preconds'] }}
{% endif %}

**Scenario:**

{% for case_step in case['custom_steps_separated'] %}

*Step {{ loop.index }}.* {{ case_step['content'] }}

*Expected result:*

{{ case_step['expected'] }}

{% endfor %}
{% endif %}
```

You can use all parameters of two variables in the template — *case* and *params*. Case parameters depends on TestRail template. All custom parameters have prefix "custom_" before system name set in TestRail.

Here is an example of *case* variable (parameters depends on case template):

```
case = {
    'created_by': 3,
    'created_on': 1524909903,
    'custom_expected': None,
    'custom_goals': None,
    'custom_mission': None,
    'custom_preconds': '- The user is not registered in the
system.\r\n'
        '- Registration form opened.',
    'custom_steps': '',
    'custom_steps_separated': [{
        'content': 'Enter mobile phone number.',
        'expected': '- Entered phone number '
        'is visible in the form field.'
        },
        {'content': 'Press OK button.',
        'expected': '- SMS with registration code '
        'received.\n'}],
    'custom_test_androidtv': None,
    'custom_test_appletv': None,
    'custom_test_smarttv': 'None,
    'custom_tp': True,
    'estimate': None,
    'estimate_forecast': None,
    'id': 15940,
    'milestone_id': None,
    'priority_id': 4,
    'refs': None,
    'section_id': 3441,
    'suite_id': 101,
    'template_id': 7,
    'title': 'Registration by mobile phone number.',
    'type_id': 7,
    'updated_by': 10,
    'updated_on': 1528978979
}
```

And here is an example of *params* variable (parameters are always the same):

```
params = {
    'multi_param_name': 'platform',
    'multi_param_sys_name': 'custom_platform',
    'multi_param_select': ['android', 'ios'],
    'multi_param_select_type': any,
    'add_cases_without_multi_param': False,
    'checkbox_param_name': 'publish',
    'checkbox_param_sys_name': 'custom_publish',
    'checkbox_param_select_type': 'checked',
    'choose_priorities': ['critical', 'high', 'medium'],
    'add_multi_param_to_case_header': True,
    'add_multi_param_to_std_table': True,
    'add_priority_to_case_header': True,
    'add_priority_to_std_table': True,
    'add_case_id_to_case_header': False,
    'add_case_id_to_std_table': False
}
```

Next three fields are necessary due localization issues. While markdown document with test cases is composed on the go, you have to set up some document headers. Definitely not the best solution in my life.

`section_header` First level header of section with test cases. By default it's *Testing program* in Russian.

`add_std_table` You can exclude (*false*) a testing table from the document.

`std_table_header` First level header of section with test results table. By default it's *Testing table* in Russian.

`std_table_column_headers` Semicolon separated headers of testing table columns. By default it's *№; Priority; Platform; ID; Test case name; Result; Comment* in Russian.

`add_suite_headers` With *false* you can exclude all suite headers from the final document.

`add_section_headers` With *false* you can exclude all section headers from the final document.

`add_case_id_to_case_header` Every test case in TestRail has unique ID, which, as usual, you can find in the header or test case URL: http://testrails.url/index.php?/cases/view/15920... <-. So you can add (*true*) this ID to the test case headers and testing table. Or not (*false*).

`add_case_id_to_std_table` Also you can add (*true*) the column with the test case IDs to the testing table.

In TestRail you can add custom parameters to your test case template. With next settings you can use one *multi-select* or *dropdown* (good for platforms, for example) and one *checkbox* (publishing) plus default *priority* parameter for cases sampling.

`multi_param_name` Parameter name of *multi-select* or *dropdown* type you set in *System Name* field of *Add Custom Field* form in TestRail. For example, *platforms* with values *Android*, *iOS*, *PC*, *Mac* and *web*. If *multi_param_select* not set, all test cases will be downloaded (useful when you need just to add parameter value to the test headers or testing table).

`multi_param_select` Here you can set the platforms for which you want to get test cases (case insensitive). For example, you have similar UX for mobile platforms and want to combine them:

```
preprocessors:
  - testrail:
    ...
    multi_param_name: platforms
    multi_param_select: android, ios
    ...
```

`multi_param_select_type` With this parameter you can make test cases sampling in different ways. It has several options:

- *any* (by default) — at least one of *multi_param_select* values should be set for the case,
- *all* — all of *multi_param_select* values should be set and any other can be set for the case,
- *only* — only *multi_param_select* values in any combination should be set for the case,
- *match* — all and only *multi_param_select* values should be set for the case.

With *multi_param_select: android, ios* we will get the following cases:

Test cases	Android	iOS	PC	Mac	web	any	all	only	match
Test case 1	+	+				+	+	+	+
Test case 2	+	+				+	+	+	+
Test case 3			+	+					
Test case 4		+	+	+		+			
Test case 5	+	+			+	+	+		
Test case 6	+	+			+	+	+		
Test case 7			+	+	+				

Test cases	Android	iOS	PC	Mac	web	any	all	only	match
Test case 8			+	+	+				
Test case 9		+				+		+	

`add_cases_without_multi_param` Also you can include (by default) or exclude (*false*) cases without any value of *multi-select* or *dropdown* parameter.

`add_multi_param_to_case_header` You can add (*true*) values of *multi-select* or *dropdown* parameter to the case headers or not (by default).

`add_multi_param_to_std_table` You can add (*true*) column with values of *multi-select* or *dropdown* parameter to the testing table or not (by default).

`checkbox_param_name` Parameter name of *checkbox* type you set in *System Name* field of *Add Custom Field* form in TestRail. For example, *publish*. Without parameter name set all of cases will be downloaded.

`checkbox_param_select_type` With this parameter you can make test cases sampling in different ways. It has several options:

- *checked* (by default) — only cases whith checked field will be downloaded,
- *unchecked* — only cases whith unchecked field will be downloaded.

`choose_priorities` Here you can set test case priorities to download (case insensitive).

```
preprocessors:
  - testrail:
    ...
    choose_priorities: critical, high, medium
    ...
```

`add_priority_to_case_header` You can add (*true*) priority to the case headers or not (by default).

`add_priority_to_std_table` You can add (*true*) column with case priority to the testing table or not (by default).

Using described setting you can flexibly adjust test cases sampling. For example, you can download only published *critical* test cases for both and only *Mac* and *PC*.

Now strange things, mostly made specially for my project, but may be useful for others.

Screenshots. There is no possibility to store screenshots in TestRail projects, but you can store them in the GitLab repository (link to which should be stated in one of the following parameters). GitLab project should have following structure:

```
images/├──
 smarttv/
|    ├── screenshot1_smarttv.png
|    ├── screenshot2_smarttv.png
|    └── ...├──
 androidtv/
|    ├── screenshot1_androidtv.png
|    ├── screenshot2_androidtv.png
|    └── ...├──
 appletv/
|    ├── screenshot1_appletv.png
|    ├── screenshot2_appletv.png
|    └── ...├──
 web/
|    ├── screenshot1_web.png
|    ├── screenshot2_web.png
|    └── ...├──
 screenshot1.png├──
 screenshot2.png└──
 ...
```

images folder used for projects without platforms.

Filename ending is a first value of *multi_param_select* parameter (*platform*). Now to add screenshot to your document just add following string to the test case (unfortunately, in TestRail interface it will looks like broken image link):

```
(leading exclamation mark here!)[Image title](
main_filename_part)
```

Preprocessor will convert to the following format:

```
https://gitlab.url/gitlab_group_name/gitlab_project_name/raw
/master/images/platform_name/
main_filename_part_platform_name.png
```

For example, in the project with *multi_param_select: smarttv* the string

```
(leading exclamation mark here!)[Application main screen](
main_screen)
```

will be converted to:

```
(leading exclamation mark here!)[Application main screen](
https://gitlab.url/documentation/application-screenshots/raw
/master/images/smarttv/main_screen_smarttv.png)
```

That's it.

`resolve_urls` Turn on (*true*) or off (*false*, by default) image urls resolving.

`screenshots_url` GitLab repository URL, in our example: https://gitlab.url/documentation/application-screenshots/ .

`screenshots_ext` Screenshots extension. Yes, it must be only one and the same for all screenshots.

And the last one emergency tool. If you have no jinja template for any type of TestRail case, you'll see this message like this: *There is no jinja template for test case template_id 5 (case_id 1325) in folder case_templates.* So you have to write jinja template by yourself. To do this it's necessary to know case structure. This parameter shows it to you.

`print_case_structure` Turn on (*true*) or off (*false*, by default) printing out of case structure with all data in it if any problem occurs.

Usage

Just add the preprocessor to the project config, set it up and enjoy the automatically collected test cases to your document.

Tips

In some cases you may encounter a problem with test cases text format, so composed markdown file will be converted to the document with bad formatting. In this cases *replace* preprocessor could be useful: https://foliant-docs.github.io/docs/preprocessors/replace/ .

CLI Extensions

Bump

This CLI extension adds bump command that lets you bump Foliant project semantic version without editing the config manually.

Installation

```
$ pip install foliantcontrib.bump
```

Usage

Bump version from “1.0.0” to “1.0.1”:

```
$ foliant bump
Version bumped from 1.0.0 to 1.0.1.
```

Bump major version:

```
$ foliant bump -v major
Version bumped from 1.0.1 to 2.0.0.
```

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

```
$ foliant bump --help
usage: foliant bump [-h] [-v VERSION_PART] [-p PATH] [-c
CONFIG]

Bump Foliant project version.

optional arguments:
  -h, --help            show this help message and exit
  -v VERSION_PART, --version-part VERSION_PART
```

```
                        Part of the version to bump: major,
minor, patch, prerelease, or build (default: patch).
  -p PATH, --path PATH  Path to the directory with the
config file (default: ".").
  -c CONFIG, --config CONFIG
                        Name of the config file (default: "
foliant.yml").
```

Gupload

Gupload is the Foliant CLI extension, it's used to upload created documents to Google Drive.

Gupload adds `gupload` command to Foliant.

Installation

```
$ pip install foliantcontrib.gupload
```

Config

To config the CLI extension, add `gupload` section in the project config. As `gupload` needs document to upload, appropriate backend settings also have to be here.

CLI extension has a number of options (all fields are required but can have no values):

```
gupload:
    gdrive_folder_name: Foliant upload
    gdrive_folder_id:
    gdoc_title:
    gdoc_id:
    convert_file:
    com_line_auth: false
```

`gdrive_folder_name` Folder with this name will be created on Google Drive to upload file.

`gdrive_folder_id` This field is necessary to upload files to previously created folder.

`gdoc_title` Uploaded file will have this title. If empty, real filename will be used.

`gdoc_id` This field is necessary to rewrite previously uploaded file and keep the link to it.

`convert_file` Convert uploaded files to google docs format or not.

`com_line_auth` In some cases it's impossible to authenticate automatically (for example, with Docker), so you can set *True* and use command line authentication procedure.

Usage

At first you have to get Google Drive authentication file.

1. Go to APIs Console and make your own project.
2. Go to library, search for "Google Drive API", select the entry, and click "Enable".
3. Select "Credentials" from the left menu, click "Create Credentials", select "OAuth client ID".
4. Now, the product name and consent screen need to be set -> click "Configure consent screen" and follow the instructions. Once finished:
   - Select "Application type" to be *Other types*.
   - Enter an appropriate name.
   - Input http://localhost:8080 for "Authorized JavaScript origins".
   - Input http://localhost:8080/ for "Authorized redirect URIs".
   - Click "Save".
5. Click "Download JSON" on the right side of Client ID to download client_secret_-.json. The downloaded file has all authentication information of your application.
6. Rename the file to "client_secrets.json" and place it in your working directory near foliant.yml.

Now add the CLI extension to the project config with all settings strings. At this moment you have no data to set *Google Drive folder ID* and *google doc ID* so keep it empty.

Run Foliant with `gupload` command:

```
$ foliant gupload docx✔
 Parsing config✔
 Applying preprocessor flatten✔
 Making docx with Pandoc──────────────────────

```

```
Result: filename.docx─────────────────────✔

 Parsing config
Your browser has been opened to visit:

    https://accounts.google.com/o/oauth2/auth?...

Authentication successful.✔
 Uploading 'filename.docx' to Google Drive
─────────────────────

Result:
Doc link: https://docs.google.com/document/d/1
GPvNSMJ4ZutZJwhUYM1xxCKWMU5Sg/edit?usp=drivesdk
Google drive folder ID: 1AaiWMNIYlq9639P30R3T9
Google document ID: 1GPvNSMJ4Z19YM1xCKWMU5Sg
```

Authentication form will be opened. Choose account to log in.

With command line authentication procedure differs little bit:

```
$ docker-compose run --rm foliant gupload docx✔
 Parsing config✔
 Applying preprocessor flatten✔
 Making docx with Pandoc─────────────────────

Result: filename.docx─────────────────────✔

 Parsing config
Go to the following link in your browser:

    https://accounts.google.com/o/oauth2/auth?...

Enter verification code: 4/XgBllTXpxv8kKjsiTxLc
Authentication successful.✔
 Uploading 'filename.docx' to Google Drive
─────────────────────
```

```

Result:
Doc link: https://docs.google.com/document/d/1
GPvNSMJ4ZutZJwhUYM1xxCKWMU5Sg/edit?usp=drivesdk
Google drive folder ID: 1AaiWMNIYlq9639P30R3T9
Google document ID: 1GPvNSMJ4Z19YM1xCKWMU5Sg
```

Copy link from terminal to your browser, choose account to log in and copy generated code back to the terminal.

After that the document will be uploaded to Google Drive and opened in new browser tab.

Now you can use *Google Drive folder ID* to upload files to the same folder and *google doc ID* to rewrite document (also you can IDs in folder and file links).

Notes

If you set up *google doc ID* without *Google Drive folder ID* file will be moved to the new folder with the same link.

Meta

pypi v1.1.0

Metadata for Foliant

This extension adds the `meta generate` command to Foliant, which generates the yaml-file with project metadata. It also allows to add other meta commands `meta <command>` which use the generated metadata.

It also adds the preprocessor `meta` which removes metadata blocks from the documents before builds and allows inserting formatted strings on the place of meta blocks, based on specific metadata keys.

Installation

```
$ pip install foliantcontrib.meta
```

Specifying metadata

Metadata may be specified in the beginning of a Markdown-file using YAML Front Matter format:

```
---
id: MAIN_DOC
title: Description of the product
key: value
---
```

`meta generate` command

Usage

To generate meta file run the `meta generate` command:

```
$ foliant meta generate
```

Metadata for the document will appear in the `meta.yml` file.

Config

Meta generate command has just one option right now. It is specified under `meta` section in config:

```
meta:
    filename: meta.yml
```

`filename` name of the YAML-file with generated project metadata.

`meta` preprocessor

`meta` preprocessor allows you to remove metadata from your Markdown source files before build. It may be necessary if some backend doesn't accept the YAML Front Matter syntax.

This preprocessor also offers you a feature which we call *seeds*:

Seeds are little string templates which will may be used to add some text after the metadata block in the resulting document, if specific keys were mentioned in the metadata. Details in the **Seeds** section.

Usage

Add `meta` preprocessor to your `preprocessors` section of foliant.yml and specify all your seeds:

```
preprocessors:
    - meta:
        delete_meta: true
        seeds:
            section: '*Section "{value}"*'
            id: <anchor>{value}</anchor>
```

`delete_meta` If set to `true` — metadata block will be deleted from the document before build. Default: `false`

`seeds` Seeds dictionary. Details in the next section.

Seeds

Seeds allow you to add small chunks of text based on specific keys mentioned in the metadata block. For example, if you wish to add a small subcaption at the beginning of the document, which will use this document's title, add the `title` seed:

```
preprocessors:
    - meta:
        seeds:
            title: '*Section "{value}"*'
```

If we have a meta block like this in our document:

```
---
ID: legal_info
relates: index.md
```

```
title: Legal information
---

### Terms of use
```

Preprocessor will notice that `title` key was used in the meta block, and will add the seed with `{value}` placeholder replaced by the value of the `title` field:

```
*Section "Legal information"

### Terms of use
```

ProjectGraph

pypi v1.0.1

ProjectGraph

Foliant Meta Command which draws a scheme of project sections. This extension uses meta-information, collected by folinatcontrib.meta extension.

Graphviz is used to build a scheme.

`libgraphviz-dev` is required to be installed on your machine.

Installation

```
$ pip install foliantcontrib.project_graph
```

Usage

First you need to specify all relations between the documents in your project. To do this add the `relates` section to your document's meta-data:

```
---
relates:
    - tests/test1.md
    - specs/spec.md
---
```

in `relates` section you need to specify a list of documents to which current document relates. You can specify either a relative path to connected document or its ID (if the corresponding document has an ID assigned in its meta section):

```
### index.md
---
id: index
---
```

```
### glossary.md
---
relates:
    - index
---
```

After you specified all relations, run the draw command:

```
$ foliant draw
```

Scheme will appear in the file `project_graph.png`

Config

ProjectGraph has a number of options:

```
project_graph:
    directed: false
    filename: project_graph.png
    gv_attributes:
        node:
            shape: rect
            color: green
        edge:
            arrowhead: open
        graph:
            ranksep: 1
        main_relation:
            penwidth: 2
```

directed Specifies graph to be directed or not. Default: `false`

filename Graph output filename. Default: `project_graph.png`

gv_attributes A dictionary with global attributes of the graph. Each dictionary should be stored under the Graphviz Entity key (`node`, `edge`, or `graph`), or under type key. All sections or relations which have this type will get these attributes.

If you want to adjust the look of just one node, add a `gv_attributes` option into the meta of the document:

```
---
id: index
relates:
    - glossary
gv_attributes:
    color: green
    shape: circle
---
```

You can also change the look of the edges, which connect nodes. To do this you can use a detailed syntax of relations.

Relations detailed syntax

As stated in the beginning, to specify relations you need to add a `relates` param and include a list of related documents IDs\file paths:

```
---
```

```
relates:
    - doc1.md
    - MAIN_SPEC
---
```

But there's also a detailed syntax for specifying relations, it looks like this:

```
---
relates:
    - rel_path: doc1.md
      type: details
    - rel_id: MAIN_SPEC
      gv_attributes:
        color: #CCCCCC
        arrowhead: none
---
```

In the detailed syntax each relation is not a string, but a mapping. This time you have to explicitly use either `rel_path` key, if you are pointing to a document by path, or `rel_id` if you do it by ID.

Also you can specify relation type by adding a `type` key. Right now the value of this key just goes to the edge label, but soon you'll be able to change the appearance of all edges with one type.

Finally you can override this specific edge's appearance by adjusting Graphviz attributes in the `gv_attributes` key.

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 “base” 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 `base`. It's used by default if no template is specified.

Init Templates

Preprocessor

Template for a Foliant preprocessor. Instead of looking for an existing preprocessor, cloning it, and modifying its source, install this package and generate a preprocessor directory. As simple as:

```
$ foliant init -t preprocessor
```

Installation

```
$ pip install --no-compile foliantcontrib.templates.
preprocessor
```

Usage

```
$ foliant init -t preprocessor
Enter the project name: Awesome Preprocessor✔
 Generating project─────────────────────

Project "Awesome Preprocessor" created in awesome-
preprocessor
```

Or:

```
$ foliant init -t preprocessor -n "Awesome Preprocessor"✔
 Generating project─────────────────────

Project "Awesome Preprocessor" created in awesome-
preprocessor
```

Result:

```
$ tree awesome-preprocessor
.├──
 changelog.md├──
 foliant│
    └── preprocessors│
        └── awesome-preprocessor.py├──
 LICENSE├──
 README.md└──
 setup.py

2 directories, 5 files
```

Src

This extension supports the command `src` to backup the source directory of Foliant project (usually called as `src`) and to restore it from prepared backup.

Backing up of the source directory is needed because MultiProject extension modifies this directory by moving the directories of built subprojects into it.

Installation

To enable the `src` command, install MultiProject extension:

```
$ pip install foliantcontrib.multiproject
```

Usage

To make a backup of the source directory, use the command:

```
$ foliant src backup
```

To restore the source directory from the backup, use the command:

```
$ foliant src restore
```

You may use the `--config` option to specify custom config file name of your Foliant project. By default, the name `foliant.yml` is used:

```
$ foliant src backup --config alternative_config.yml
```

Also you may specify the root directory of your Foliant project by using the `--project-dir` option. If not specified, current directory will be used.

Subset

This CLI extension adds the command `subset` that generates a config file for a subset (i.e. a detached part) of the Foliant project. The command uses:

- the common (i.e. default, single) config file for the whole Foliant project;
- the part of config that is individual for each subset. The Foliant project may include multiple subsets that are defined by their own partial config files.

The command `subset` takes a path to the subset directory as a mandatory command line parameter.

The command `subset`:

- reads the partial config of the subset;
- optionally rewrites the paths of Markdown files that specified there in the `chapters` section;
- merges the result with the default config of the whole Foliant project config;
- finally, writes a new config file that allows to build a certain subset of the Foliant project with the `make` command.

Installation

To install the extension, use the command:

```
$ pip install foliantcontrib.subset
```

Usage

To get the list of all necessary parameters and available options, run `foliant subset --help`:

```
$ foliant subset --help
usage: foliant subset [-h] [-p PROJECT_DIR_PATH] [-c CONFIG]
 [-n] [-d] SUBPATH

Generate the config file to build the project subset from
SUBPATH.

positional arguments:
  SUBPATH               Path to the subset of the Foliant
project

optional arguments:
  -h, --help            show this help message and exit
  -p PROJECT_DIR, --project-dir PROJECT_DIR
                        Path to the Foliant project
  -c CONFIG, --config CONFIG
                        Name of config file of the Foliant
project, default 'foliant.yml'
  -n, --no-rewrite      Do not rewrite the paths of Markdown
 files in the subset partial config
  -d, --debug           Log all events during build. If not
set, only warnings and errors are logged
```

In most cases it's enough to use the default values of optional parameters. You need to specify only the `SUBPATH`—the directory that should be located inside the Foliant project source directory.

Suppose you use the default settings. Then you have to prepare:

- the common (default) config file `foliant.yml` in the Foliant project root directory;
- partial config files for each subset. They also must be named `foliant.yml`, and they must be located in the directories of the subsets.

Your Foliant project tree may look so:

```
$ tree
.├──
 foliant.yml└──
 src
    ├── group_1
    │   ├── product_1
    │   │   └── feature_1
    │   │       ├── foliant.yml
    │   │       └── index.md
    │   └── product_2
    │       ├── foliant.yml
    │       └── main.md
    └── group_2
        ├── foliant.yml
        └── intro.md
```

The command `foliant subset group_1/product_1/feautre_1` will merge the files `./src/group_1/product_1/feautre_1/foliant.yml` and `./foliant.yml`, and write the result into the file `./foliant.yml.subset`.

After that you may use the command like the following to build your Foliant project:

```
$ foliant make pdf --config foliant.yml.subset
```

Let's look at some examples.

The content of the common (default) file `./foliant.yml`:

```
title: &title Default Title

subtitle: &subtitle Default Subtitle

version: &version 0.0

backend_config:
    pandoc:
        template: !path /somewhere/template.tex
        reference_docx: !path /somewhere/reference.docx
        vars:
            title: *title
            version: *version
            subtitle: *subtitle
            year: 2018
        params:
            pdf_engine: xelatex
```

The content of the partial config file `./src/group_1/product_1/feautre_1/foliant.yml`:

```
title: &title Group 1, Product 1, Feature 1

subtitle: &subtitle Technical Specification
```

```

version: &version 1.0

chapters:
    - index.md

backend_config:
    pandoc:
        vars:
            year: 2019
```

The content of newly generated file `./foliant.yml.subset`:

```
title: &title Group 1, Product 1, Feature 1
subtitle: &subtitle Technical Specification
version: &version 1.0
backend_config:
    pandoc:
        template: !path /somewhere/template.tex
        reference_docx: !path /somewhere/reference.docx
        vars:
            title: *title
            version: *version
            subtitle: *subtitle
            year: 2019
        params:
            pdf_engine: xelatex
chapters:
- b2b/order_1/feature_1/index.md
```

If the option `--no-rewrite` is not set, the paths of Markdown files that are specified in the `chapters` section of the file `./src/group_1/product_1/feautre_1/foliant.yml`, will be rewritten as if these paths were relative to the directory `./src/group_1/product_1/feautre_1/`.

Otherwise, the Subset CLI extension will not rewrite the paths of Markdown files as if they were relative to `./src/` directory.

Note that the Subset CLI Extension merges the data of the config files recursively, so any subkeys of default config may be overridden by the settings of partial config.

Config Extensions

MultiProject

This extension resolves the `!from` YAML tag in the project config and replaces the value of the tag with `chaptres` section of related subproject.

Installation

```
$ pip install foliantcontrib.multiproject
```

Usage

The subproject location may be specified as a local path, or as a Git repository with optional revision (branch name, commit hash or another reference).

Example of `chapters` section in the project config:

```
chapters:
    - index.md
    - !from local_dir
    - !from https://github.com/foliant-docs/docs.git
    - !from https://github.com/some_other_group/
some_other_repo.git#develop
```

Before building the documentation superproject, Multiproject extension calls Foliant to build each subproject into `pre` target, and then moves the directories of built subprojects into the source directory of the superproject (usually called as `src`).

Note that Foliant allows to override default config file name `foliant.yml` by using `--config` or `-c` command line option. To provide correct working of Multiproject extension, the same names of config files should be used in the superproject and in all subprojects.

Slugs

Slugs is an extension for Foliant to generate custom slugs from arbitrary lists of values.

It resolves `!slug`, `!date`, `!version`, and `!commit_count` YAML tags in the project config.

The list of values after the `!slug` tag is replaced with the string that joins these values using `-` delimeter. Spaces ( ) in the values are replaced with underscores (`_`).

The value of the node that contains the `!date` tag is replaced with the current local date.

The list of values after the `!version` tag is replaced with the string that joins these values using `.` delimeter.

The value of the node that contains the `!commit_count` tag is replaced by the number of commits in the current Git repository.

Installation

```
$ pip install foliantcontrib.slugs
```

Usage

Slug

Config example:

```yaml
title: &title My Awesome Project
version: &version 1.0
slug: !slug
    - *title
    - *version
    - !date
```

Example of the resulting slug:

```
My_Awesome_Project-1.0-2018-05-10
```

Note that backends allow to override the top-level slug, so you may define different custom slugs for each backend:

```
backend_config:
    pandoc:
        slug: !slug
            - *title
            - *version
            - !date
    mkdocs:
        slug: my_awesome_project
```

Version

Config example:

```
version: !version [1, 0, 5]
```

Resulting version:

```
1.0.5
```

If you wish to use the number of commits in the current branch as a part of your version, add the `!commit_count` tag:

```
version: !version
    - 1
    - !commit_count
```

Resulting version:

```
1.85
```

The `!commit_count` tag accepts two arguments:

- name of the branch to count commits in;
- correction—a positive or negative number to adjust the commit count.

Suppose you want to bump the major version and start counting commits from the beginning. Also you want to use only number of commits in the `master` branch. So your config will look like this:

```
version: !version
    - 2
    - !commit_count master -85
```

Result:

```
2.0
```

History of Releases

Here is the single linear history of releases of Foliant and its extensions. It’s also available as an RSS feed.