

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](http://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 custom 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

### 1.0.11

- Allow to specify custom options for EscapeCode preprocessor as the `escape_code.options` config parameter value.
- Pass the `quiet` flag to `BaseParser()` as an optional argument for using in config extensions.

### 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 it 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
```

And finally, install the Pandoc backend:

```
$ python3 -m pip install foliantcontrib.pandoc
```

## 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
```

And finally, install the Pandoc backend:

```
$ python3 -m pip install foliantcontrib.pandoc
```

## 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:

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

2. Install Foliant with pip:

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

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

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

And finally, install the Pandoc backend:

```
$ python3 -m pip install foliantcontrib.pandoc
```

## 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:

```
1 $ foliant init
2 Enter the project name: Hello Foliant✓
3 Generating Foliant project-----
4
5 Project "Hello Foliant" created in hello-foliant
```

To do the same with Docker, run:

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

Here's what this command created:

```
1 $ cd hello-foliant
2 $ tree
3 .
4   ├── docker-compose.yml
5   ├── Dockerfile
6   ├── foliant.yml
7   ├── README.md
8   └── requirements.txt
```

```
9  src
10    └── index.md
11
12 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 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

To build a site you will first need a suitable backend (to catch up with the terminology, check [this article](#)). Let's start with **MkDocs** backend. First, install it using the following command (or skip to the docker example):

```
python3 -m pip install foliantcontrib.mkdocs
```

Then, in the project directory, run:

```
1 $ foliant make site
2 Parsing config... Done
3 Applying preprocessor mkdocs... Done
4 Applying preprocessor _unescape... Done
5 Making site with MkDocs... Done—————
6
7 Result: Hello_Foliant-2020-05-25.mkdocs
```

Or, with Docker Compose:

```
1 $ docker-compose run --rm foliant make site
2 Parsing config... Done
3 Applying preprocessor mkdocs... Done
4 Applying preprocessor _unescape... Done
5 Making site with MkDocs... Done—————
```

6

7 Result: Hello\_Foliant-2020-05-25.mkdocs

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

```
1 $ python3 -m http.server -d Hello_Foliant-2020-05-25.mkdocs
2 Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Open [localhost:8000](http://localhost:8000) in your web browser. You should see something like this:



**Figure 1.** Basic Foliant project built with MkDocs

## Build PDF

### Note

To build PDF with Pandoc, make sure you have it and TeXLive installed (see [Installation](#)).

In the project directory, run:

```
1 $ foliant make pdf
2 Parsing config... Done
3 Applying preprocessor flatten... Done
```

```
4 Applying preprocessor _unescape... Done
5 Making pdf with Pandoc... Done————
6
7 Result: Hello_Foliant-2020-05-25.pdf
```

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

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

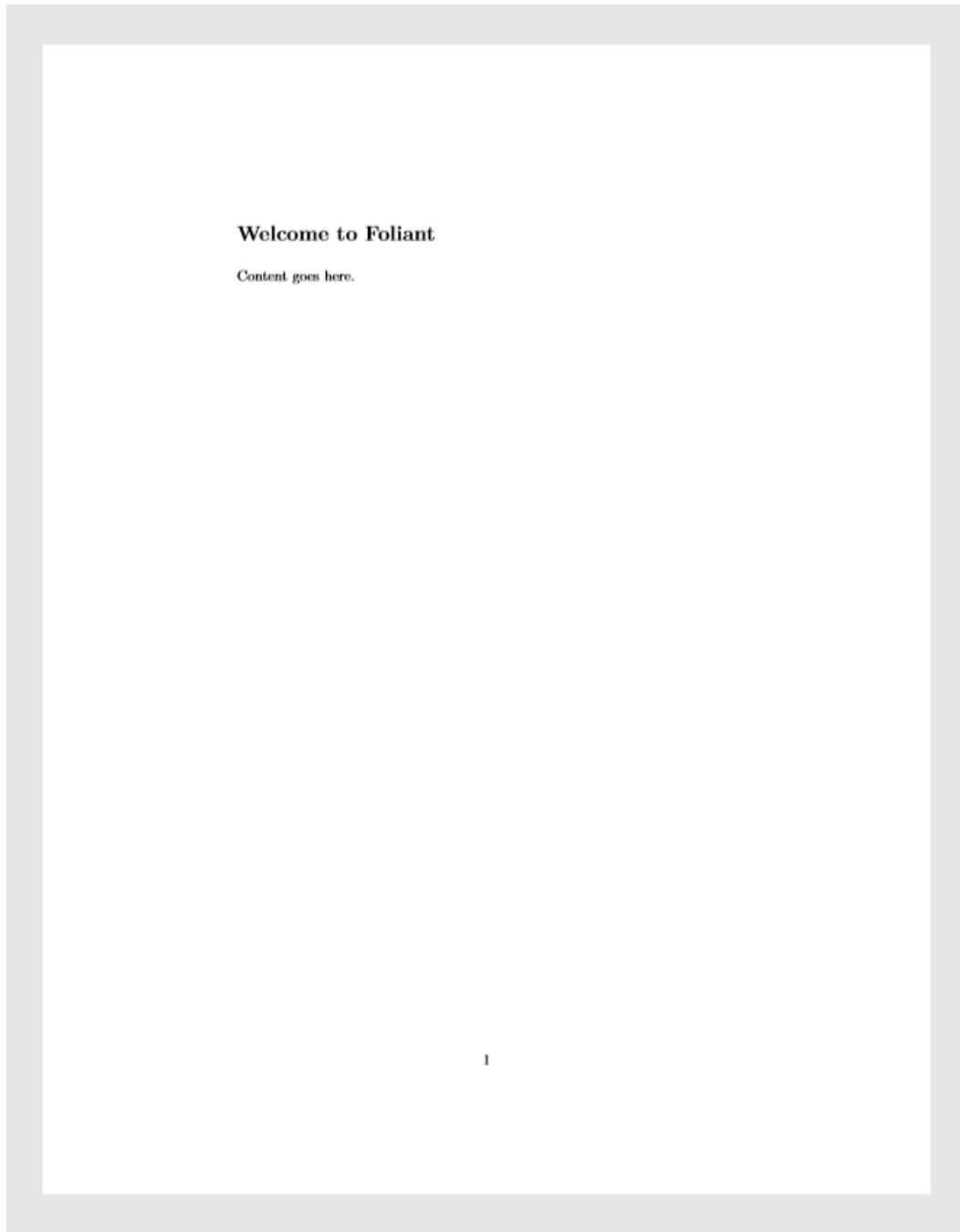
### Note

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

```
1 $ docker-compose run --rm foliant make pdf
2 Parsing config... Done
3 Applying preprocessor flatten... Done
4 Applying preprocessor _unescape... Done
5 Making pdf with Pandoc... Done————
6
7 Result: Hello_Foliant-2020-05-25.pdf
```

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



**Figure 2.** Basic Foliant project built with Pandoc

# Edit Content

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

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

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

```
1 # Hello Again  
2  
3 This is regular text generated from regular Markdown.  
4  
5 Foliant 'doesnt force any *special* Markdown flavor.
```

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

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

Rebuild the project to see the new page:

```
1 $ docker-compose run --rm foliant make site && docker-  
  compose run --rm foliant make pdf  
2 Parsing config... Done  
3 Applying preprocessor mkdocs... Done  
4 Applying preprocessor _unescape... Done  
5 Making site with MkDocs... Done—————  
6  
7 Result: Hello_Foliant-2020-05-25.mkdocs  
8 Parsing config... Done  
9 Applying preprocessor flatten... Done  
10 Applying preprocessor _unescape... Done  
11 Making pdf with Pandoc... Done—————  
12  
13 Result: Hello_Foliant-2020-05-25.pdf
```

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



**Figure 3.** New page on the site



**Figure 4.** New page in the pdf document

# Use Preprocessors

Preprocessors is what makes Foliant special and extremely flexible. 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.

First, we need to install the `blockdiag` preprocessor:

```
$ python3 -m pip install foliantcontrib.blockdiag
```

Or, if you are building with docker, add `foliantcontrib.blockdiag` to `requirements.txt` and rebuild the image with `docker-compose build` command.

Next, we need to switch on the `blockdiag` preprocessor in config. Open `foliant .yml` and add the following lines:

```
1 title: Hello Foliant  
2 +
```

```
3 + preprocessors:  
4 +   - blockdiag  
5  
6 chapters:  
7   - index.md  
8   - hello.md
```

Then, in `hello.md`, add the following:

```
1 Foliant 'doesn't force any *special* Markdown flavor.  
2  
3 + <>seqdiag caption="This diagram is generated on the fly">  
4 +   seqdiag {  
5 +     "foliant make site" -> "mkdocs processor" -> "  
6     blockdiag processor" -> "mkdocs backend" -> site;  
7 +   }  
7 + </seqdiag>
```

Blockdiag processor 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 Foliant   Welcome to Foliant   Hello Again   Q Search   ← Previous   Next →

[Hello Again](#)  
[About Foliant](#)

# Hello Again

This is regular text generated from regular Markdown.  
Foliant doesn't force any special Markdown flavor.

```

sequenceDiagram
    participant A as "foliant make site"
    participant B as "mkdocs preprocessor"
    participant C as "blocking preprocessor"
    participant D as "mkdocs backend"
    participant E as "site"
    A->>B: 
    activate B
    B->>C: 
    activate C
    C->>D: 
    activate D
    D->>E: 
    deactivate D
    deactivate C
    deactivate B
  
```

**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 is there too:



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`:

```
1 preprocessors:  
2 - - blockdiag  
3 + - blockdiag:  
4 +     params:  
5 +         font: !path comic.ttf
```

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



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

## Hello Again

This is regular text generated from regular Markdown.

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

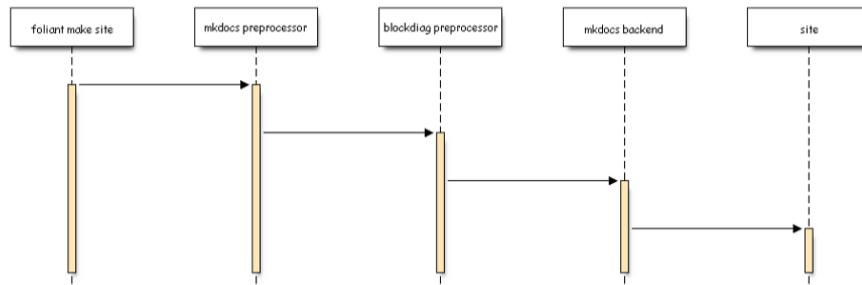


Figure 1: This diagram is generated on the fly

**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 particular 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:

```
1 This is a diagram that is rendered to `*.png` in html and to
  `*.pdf` in pdf:
2
3 <<blockdiag format="<<if targets="html">png</if><<if targets
  ="pdf">pdf</if">
4     ...
5 </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.

# Architecture And Basic Design Concepts

Foliant is an open-source software written in Python.

Foliant has modular architecture. Foliant itself, also known as [Foliant Core](#), is a relatively compact and rarely updated package. Foliant Core centrally manages extensions and provides base classes for developing extensions according to a common pattern.

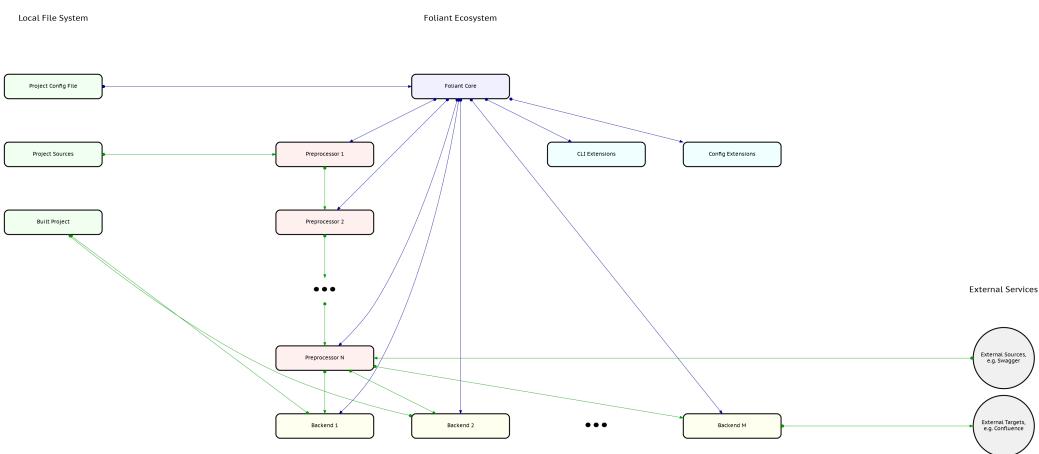
Suddenly, Foliant Core doesn't build documentation projects. All application functionality is provided by Foliant extensions. There are almost 50 extensions for Foliant. If you don't need some extensions, you may not to install them.

There are 4 types of Foliant extensions.

- **Backends** are modules that prepare your Markdown content in accordance to requirements of third-party software that builds some final **targets**, e.g. PDF files or static sites. Also backends manage external commands that call such software. Usually one backend manages one external command, for example, `pandoc` or `mkdocs`. A backend may not to call any external command but upload your content to some external service, e.g. Confluence. A single backend may generate multiple targets. Different backends may build the same target. For example, the target `site` (a static site) can be built with 3 backends: MkDocs, Slate, and Aglio. If more than one of them are installed, you should choose the certain backend to build the required target: it may be specified in the command line or asked interactively.
- **Preprocessors** are modules that perform various transformations of your source Markdown content before passing the content to a backend. Preprocessors are the most numerous type of Foliant extensions. They may be very simple like [RemoveExcess](#), or pretty complicated like [Includes](#). Each Foliant project may use any number of preprocessors that are applied sequentially, one after another. Preprocessors provide many kinds of content transformations. Preprocessors may:
  - replace something in your content according to some rules;
  - draw diagrams and schemes from code;
  - include content from external files into your sources;
  - get data for your documentation project from external services, e.g. remote Git repositories, Swagger, Testrail, Figma, Sympli, etc.;

- set high-level semantic relations between different parts of your content to provide smart cross-target links, or to restructure your single-source documentation automatically, context-dependently;
- and even run arbitrary external commands that can do anything with your files.
- **CLI extensions** extend Foliant’s command-line interface and provide additional actions that may be called from the command line.
- **Config extensions** provide additional actions that are performed during reading the project’s config. For example, MultiProject extension allows to include multiple nested Foliant projects into a parent Foliant project’s structure.

Foliant’s architecture is shown at the following diagram.



**Figure 9.** Architecture of Foliant

# Project Configuration

Configuration for Foliant is kept in a [YAML](#)-file in the project root. The default filename is `foliant.yml` (and that's how it is referred to in other sections of this site), but you can pick a different name. In this case you should run foliant with `--config` option:

```
$ foliant make pdf --config myconf.yml
```

## Config Sections

`foliant.yml` consists of several sections:

- root options,
- chapters,
- preprocessors,
- `backend_config`.

## Root Options

These are the options that are placed at the root of the config file. There are several built-in options, which are described below, but extensions may introduce their own root options (for example, [AltStructure](#) or [EscapeCode](#)), so please refer to each extension's respective docs for details.

Here are all built-in root options:

```
1 title: My Awesome Project  
2 slug: myproj  
3 src_dir: src  
4 tmp_dir: __folianttmp__
```

**title (string)** Project title. It will be used to generate the resulting file name, if `slug` option is not defined.

**slug (string)** Slug is a string which will be used to name the output file or folder after build. For example, if `slug` is `myproj`, after building PDF with foliant, the output will be saved in `myproj.pdf`. If not defined – `title` will be used to generate filename.

**src\_dir (string)** Name of the directory with your project's Markdown source files.

Normally you wouldn't want to set this option to something other than default.

Default: `src`.

**tmp\_dir (string)** Name of the directory where the intermediate files will be stored

during preprocessor pipeline execution. Normally you wouldn't want to set this option to something other than default. Default: `__folianttmp__`.

## chapters

(list)

`chapters` is a list of paths to the Markdown sources which you want to be used in the project. These paths are always relative to your `src` dir.

Here's a basic chapters list:

```
1 chapters:  
2   - intro.md  
3   - definitions.md  
4   - tutorial.md
```

Chapters may be not just a flat list, but may contain mappings and sublists. These complex structures may be treated differently by different backends: some backends may ignore mappings, some may use them to alter the resulting build. But usually these two ideas are shared between all backends:

- only those Markdown files which are mentioned in the chapters list, will appear in the resulting build;
- the order in which chapters are mentioned in the list, will be preserved in the resulting build.

Consider this example chapters list:

```
1 chapters:  
2   - intro.md  
3   - definitions.md  
4   - How To Use This Tutorial: tutorial_help.md  
5   - Creating Documentation With Foliант:  
6     - prerequisites.md  
7     - Preparing Config:  
8       - root options.md  
9       - chapters.md
```

```
10          - preprocessors.md  
11          - backend_config.md  
12      - create_sources.md  
13      - building_project.md
```

In this example first two chapters are defined as a simple list, third chapter is a mapping with one element, and after that we see several mappings with nested lists.

If we were building a PDF document with [Pandoc backend](#) or a static site with [Slate backend](#), this complex chapter structure will be just ignored, as if we have supplied a simple flat list:

```
1 chapters:  
2     - intro.md  
3     - definitions.md  
4     - tutorial_help.md  
5     - prerequisites.md  
6     - root_options.md  
7     - chapters.md  
8     - preprocessors.md  
9     - backend_config.md  
10    - create_sources.md  
11    - building_project.md
```

In any case we would get a one-file PDF or a one-page site with data from listed Markdown files in the provided order.

But if we are building site with [MkDocs backend](#), mappings become meaningful.

For example, this element:

```
- How To Use This Tutorial: tutorial_help.md
```

means “take the source from `tutorial_help.md` but change its title to `How To Use This Tutorial`” in the sidebar.

And this element:

```
1     - Creating Documentation With Foliant:  
2         - prerequisites.md  
3         - Preparing Config:  
4             - root_options.md
```

```
5           - chapters.md  
6           - preprocessors.md  
7           - backend_config.md
```

means “create a subsection in the sidebar with a title **Creating Documentation With Foliант** and nest the `prequisites.md` chapter inside. Then nest another subsection within the first one, called **Preparing Config** and nest four other chapters inside of it”.

Please, refer to each backend’s respective docs for details on how they work with chapters.

## preprocessors

(list)

All preprocessors which you want to be used in your project, should be listed under the `preprocessors` section:

```
1 preprocessors:  
2     - macros:  
3         macros:  
4             ref: <if backends="pandoc">{pandoc}</if><if  
backends="mkdocs">{mkdocs}</if>  
5         - flags  
6         - includes  
7         - blockdiag  
8         - plantuml:  
9             params:  
10                config: !path configs/plantuml.cfg  
11            - graphviz:  
12                format: svg  
13                as_image: false  
14                params:  
15                    Gdpi: 0
```

Each preprocessor has to be put in separate list item. If you don’t need to set any options, just put the preprocessor’s name in the item (flags, includes and blockdiag in the example above). If you are setting preprocessor options, then make it a mapping,

with field name being the preprocessor name, and field value – another mapping, with preprocessor options. (macros, plantuml and graphviz in the example above).

Please, refer to each preprocessor's respective docs for details on which options they have.

There are several things you have to keep in mind when setting up the preprocessors section:

### The order matters

The order, in which the preprocessors are defined in the list, is the order they are run during build. For example, if you are using Includes preprocessor to get source code for a PlantUML scheme like this:

```
1 <plantuml>
2     <include url="http://example.com/scheme.puml"></include>
3 </plantuml>
```

then Includes must be defined before PlantUML in the list, otherwise you will get an error from PlantUML when it tries to process `<include>` tag instead of the scheme code.

Some preprocessors are especially sensitive to their position in the preprocessor list (for example, [SuperLinks](#)) and there may even be situations when you will have to put the same preprocessor in the list twice.

### Preprocessors are applied to all files

Generally, preprocessors just ignore the chapters list and apply to [all Markdown files](#) in the src dir. Usually this is not an issue, but sometimes preprocessor may spend a long time on the files, which may not even get into the resulting build.

We suggest you to keep your src dir clean and only put there files which are actually getting into the project. The other solution is to use [RemoveExcess](#) preprocessor, which removes all Markdown files, which are not mentioned in the chapters list, from the temporary directory.

## backend\_config

### (mapping)

Keep all your backend settings in `backend_config` section:

```
1 backend_config:
```

```

2     pandoc:
3         template: !path template/docs.tex
4         vars:
5             title: *title
6             subtitle: 'Users Manual'
7             logo: !path template/octopus-black-512.png
8         params:
9             pdf_engine: xelatex
10            listings: true
11
12        mkdocs:
13            use_title: true
14            use_chapters: true
15            use_headings: true
16            mkdocs.yml:
17                repo_name: foliant-docs/docs
18                theme:
19                    name: material
20                    custom_dir: !path ./theme/

```

Unlike  `preprocessors` section, `backend_config` is not a list but a mapping. Here the order in which you define backends, is not important.

Moreover, you can even skip adding a backend into `backend_config` and still be able to build a project with it. It will just mean that you are using default settings.

## Modifiers

Foliant defines several custom YAML-modifiers, some of which you have already met in the examples here.

### **!include**

The `!include` modifier allows to insert content from another YAML-file.

For example, if your chapters list has grown so big, that you want to keep it separately from the main config, you can put it into `chapters.yml` file and include it in `foliant.yml`:

```
chapters: !include chapters.yml
```

## `!path, !project_path, !rel_path`

When used in foliant.yml, `!path`, `!project_path`, `!rel_path` all do the same thing: they resolve the path to an absolute path to make sure the preprocessor or backend processes this file properly.

It is recommended, that whenever you supply a path to any file in options, to precede it with the `!path` modifier:

```
1 preprocessors:
2     - swaggerdoc:
3         spec_path: !path swagger.yml
4         environment:
5             user_templates: !path widdershins_templates
6     - plantuml:
7         params:
8             config: !path configs/plantuml.cfg
9
10 backend_config:
11     pandoc:
12         template: !path pandoc/tex_templates/main.tex
13         reference_docx: !path pandoc/docx_references/basic.
14             docx
```

Why there are three of them then, would you ask? The reason is that all [foliant tag options](#) in Markdown source files are in fact also YAML-strings, which means that you can supply a list in tag option like this:

```
1 <jinja2 vars="[a1, a2, a3]">
2 Received the variables!
3
4 {% for var in vars %}
5     I've got a var {{ var }}
6 {% endfor %}
7 </jinja2>
```

And that's where `!project_path`, `!rel_path` modifiers come in really handy. Now you can refer to a file which is sitting in the project root, no matter where inside the src dir your current file is:

```
1 Here are the contents of this project's config:  
2  
3 <include src="!project_path foliant.yml"></include>
```

By convention, all tag parameters, which accept paths to external files, are considered to be paths relative to current file. But if you want to play safe or make things more explicit, you may add the `!rel_path` tag, which ensures that the path the preprocessor will get, will be relative to current file:

```
1 Here are the contetns of the adjacent chapter:  
2  
3 <include src="!rel_path chapter2.md"></include>
```

`!path` modifier, if used in tag parameters, works the same as `!project_path` modifier: it returns the absolute path to the file, relative to project root.

# Debugging Builds

Building simple documentation projects with Foliant is usually straightforward. But Foliant and its extensions are designed to build complex projects. Foliant with its extensions is a powerful, customizable, and very flexible tool. If you understand what do you exactly want to do, and you can formalize it at the project config level, Foliant will perform your task efficiently and accurately.

The problem is that it sometimes may be difficult to configure all preprocessors and backends correctly for the first time. Some settings are pretty subtle and unobvious. The order of applying the preprocessors matters. Some preprocessors may work unexpectedly when paired with others. It may be necessary to apply some preprocessor twice—before and after some another preprocessor. Fetching data from external sources may also become a bottleneck.

Fortunately, Foliant will not ask you to diagnose problems with the car engine without opening the hood. Foliant provides advanced diagnostic facilities such as the following:

- detailed event logging in the special debug mode;
- the backend `pre` that does nothing, i.e. does not convert preprocessed Markdown into any target;
- keeping the temporary working directory that is used during builds, for further analysis.

## Notes on Docker Use

In this documentation, examples of shell commands in most cases represent the native use of Foliant, i.e. as an application that is installed directly on your local operating system.

In practice, Foliant is more commonly used with Docker. Public images of Foliant with different sets of extensions are available [at Docker Hub](#). The actual tags are:

- `slim` – minimal image of Foliant with no extensions;
- `latest` – same as `slim` but with the `foliant init` command support;
- `pandoc` – image of Foliant with Pandoc backend, Pandoc itself, and LaTeX (`texlive-full` Ubuntu package);
- `full` – most complete image of Foliant with all released extensions and dependencies required for them.

You can find Dockerfiles for each image in [this GitHub repository](#).

If you need to build PDFs with Pandoc/LaTeX, and some additional functionality is required, e.g. drawing diagrams with PlantUML, it's recommended to use the image with the tag `full`. This image is widely used, frequently updated, and well supported.

To get or update the Foliant Docker image, run the command:

```
$ docker pull foliant/foliant:full
```

It's not necessary but recommended to use [Docker Compose](#) utility to build images for Foliant projects and run containers based on them.

To prepare your Foliant project to be built within Docker container, add the files `Dockerfile` and `docker-compose.yml` into the project's "root" directory.

If you decided to use the image with the `full` tag, the content of `Dockerfile` should be the following:

```
FROM foliant/foliant:full
```

The file `docker-compose.yml` should define the `foliant` service in this way:

```
1 version: '3'  
2  
3 services:  
4   foliant:  
5     build:  
6       context: ./  
7       dockerfile: ./Dockerfile  
8     volumes:  
9       - ./:/usr/src/app/
```

For debug purposes, it's useful to add one more service, `bash`, to run containers with an interactive shell. Recommended full content of the file `docker-compose.yml` is the following:

```
1 version: '3'  
2  
3 services:  
4   foliant:  
5     build:
```

```
6      context: ./  
7      dockerfile: ./Dockerfile  
8      volumes:  
9          - .:/usr/src/app/  
10     bash:  
11         build:  
12             context: ./  
13             dockerfile: ./Dockerfile  
14             volumes:  
15                 - .:/usr/src/app/  
16             entrypoint: /bin/bash
```

To run Foliant project build, first, you have to build an image for the project:

```
$ docker-compose build
```

This command should be executed in the project's "root" directory.

The image for the project should be rebuilt after pulling a newer version of Foliant image. Also, if you use a custom `Dockerfile`, and some local files are mentioned in it, the image for the project should be rebuilt after any updates of these files.

To perform the project build with Foliant, run the command like the following:

```
$ docker-compose run --rm foliant make <target> --with <  
backend>
```

You need to specify the certain target and the certain backend. So, if you want to build PDF with Pandoc, the command should be:

```
$ docker-compose run --rm foliant make pdf --with pandoc
```

This command corresponds to the native command:

```
$ foliant make pdf --with pandoc
```

Note that by default the commands within Docker containers run as root. So, after running a container, you can get some files owned by root in your local file system. To avoid this, run commands inside Docker containers as a user with the user ID and group ID of your local user on the host machine:

```
$ docker-compose run --user="$(id -u):$(id -g)" --rm foliant  
make pdf --with pandoc
```

If you described the service `bash` in your `docker-compose.yml` file, you may run a container based on the project's image, with an interactive shell. To open shell for root, run:

```
$ docker-compose run --rm bash
```

To open shell for a user with the same user ID and group ID as your current user on the host machine:

```
$ docker-compose run --user="$(id -u):$(id -g)" --rm bash
```

All examples below will represent native commands. Add `docker-compose run --rm` or `docker-compose run --user="$(id -u):$(id -g)" --rm` to their beginnings to run the corresponding commands within Docker containers.

## Running Foliant in the Debug Mode

The usual command to build some target with some backend, `foliant make <target> --with <backend>`, runs Foliant in the regular mode. In this mode, Foliant and its extensions will log only events of the levels `critical`, `error`, and `warning`. Normally, there should not be critical cases (i.e. fatal crashes) and errors (they mean, for example, fails of some preprocessors, unavailability of some external services, etc.). Warnings are almost acceptable. Some preprocessors generate large numbers of specific warnings, and such behavior doesn't mean that something is wrong.

Foliant provides the `--debug` or `-d` command line option that enables the debug mode. In the debug mode, Foliant and its extensions also log events of the levels `debug` and `info`. How detailed the logging of such events is, depends on the implementation of a particular extension. Complex preprocessors like Includes usually log their actions in great detail. Usually the messages of the `info` level are informative: they indicate, for example, the beginning of some preprocessor's work. The messages with the `debug` level generally show the status of atomic operations, e.g. reading data from the certain file. These messages often contain the values of variables that are important in this context: paths to files, external commands that are called, etc.

The variant of the command that tells Foliant to build PDF with Pandoc in the debug mode, looks like:

```
$ foliant make pdf --with pandoc --debug
```

Log files will be written to the project’s “root” directory with names `*.log`, where `*` represents UNIX timestamps of the moments when the certain files were created. We were asked to implement optional overriding of logs location—it will be done in future release of Foliant Core. But it’s so easy to move logs to any custom place with a simple shell script; fixed location of logs is a really imaginary inconvenience!

Each log is a text file that contains a number of lines (records). Each record represents a single event and consists of 4 separated fields:

- date and time of registration of the event;
- context (“place,” module) that the event is registered in;
- event log level: one of CRITICAL, ERROR, WARNING, DEBUG, INFO;
- message text that explains the essence of the event.

For example, the first record of a log is usually looks like that:

```
2020-06-25 09:40:54,419 |           flt |     INFO |  
Build started.
```

The string `flt` in the second field means Foliant itself, i.e. Foliant Core.

The context is hierarchical. The following record represents an event that is registered in the preprocessor `Includes` that is called from the preprocessor `Flatten`, that is called during project build with Pandoc backend.

```
2020-06-25 09:40:54,678 | flt.pandoc.flatten.includes |  
DEBUG | Processing Markdown file: /usr/src/app/  
__folianttmp__/__all__.md
```

In this example, the message text contains the path to the file that is currently processed.

In the next example, Pandoc backend logs the external command that is called to build needed target:

```
2020-06-25 09:40:54,684 |           flt.pandoc |     DEBUG |  
PDF generation command: pandoc --template="/foliant_stuff/  
pandoc_templates/tex_templates/main.tex" --output "
```

```
My_Awesome_Project-1.0-2020-06-25.pdf" --variable title="My Awesome Project" --variable version="1.0" --variable subtitle="Description Of My Awesome Project" --variable logo="/foliant_stuff/pandoc_templates/logos/logo.png" --variable year="2020" --variable title_page --variable toc --variable tof --pdf-engine=xelatex --listings -f markdown __folianttmp__/_all__.md
```

If you suspect that the command executes wrong, you can copy it and try to run directly in an interactive shell.

Detailed logging in the debug mode allows you to quickly localize problems accurate to a specific Foliant extension, specific source Markdown file, specific syntax construction, and solve them with minimal time.

## The pre Backend and Target, and Keeping the Working Directory

Every Foliant backend takes preprocessed Markdown content and passes it to an external command. So it would be nice to see what content the backend gets, and how the backend additionally modifies this content.

During build, source files of Foliant project are moved to the temporary working directory. By default, it is called `__folianttmp__/_` and located in the “root” directory of the project. Source files of the project are kept unchanged. Any transformations are applied only to the files located in the temporary working directory.

Foliant Core provides the built-in backend `pre` that does nothing. More precisely, this backend makes the `pre` target. The `pre` target is obtained simply by copying the temporary working directory to the project directory as the result of build.

The `pre` target is the content that comes after all preprocessors are applied, but before any backend of another kind than `pre` is called.

If the backend is not the location of your problem, and your problem occurs in an extension of another type (most often in a preprocessor), you don’t need to wait each time when the backend will complete to make the requested target.

The `pre` target is convenient for debugging extensions of all types excluding backends, and especially preprocessors.

To build Foliant project to the `pre` target, run the command:

```
$ foliant make pre
```

In addition to the `pre` backend and target, Foliant Core supports the `--keep_tmp` or `-k` command line option. By default, the temporary working directory is removed after the project build. But if the `--keep_tmp` or `-k` option is specified, the temporary working directory will be kept after build.

After the project build, this directory will contain the files that are modified by all preprocessors and the chosen backend.

The following command tells Foliant to build PDF with Pandoc, keeping the temporary working directory after build:

```
$ foliant make pdf --with pandoc --keep_tmp
```

Tip: if Pandoc doesn't make PDF due to errors in LaTeX markup, you can build the target `tex` and first debug the LaTeX source. Also, you may call Pandoc directly from the command line to build PDF from LaTeX source.

## Killing Two Birds With One Stone

Now you know what debugging facilities are provided by Foliant. But we strongly recommend you to make it a rule to start debugging Foliant projects with one universal shell command:

```
$ foliant make pre --debug
```

This command tells Foliant to build the `pre` target in the debug mode. And this is a very effective way to get closer to understanding what is wrong with your project.

# Tutorials

## Working Full Foliant Docker image

In this tutorial, we will go through the steps of working with Foliant using the Full Docker image. This is the recommended way of working with Foliant, and here's why.

Internally Foliant is not an independent documentation builder, which does everything itself, but rather an orchestrator of different tools. Foliant configures and runs them under the hood as a part of its pipeline, saving you the need to run them yourself. Of course, that's not all Foliant does, it has a lot to offer in terms of text processing itself, but many preprocessors and almost all backends use external tools in one way or another.

That was our goal from the beginning: why build your own static site generator if there are already such beautiful libraries like MkDocs or Slate? Why spend a year on creating your own Markdown to PDF generator if Pandoc has already perfected this craft? And if you are not satisfied with Pandoc, you can plug into Foliant something else, like MdToPdf. This approach comes with some disadvantages though. Let me illustrate that with an example.

One of the popular ways to use Foliant is to build a static site with [API documentation](#). After making all necessary configuration you just run the command:

```
$ foliant make site -w slate
```

...and get a folder with a generated static site.

While the command is pretty simple, internally Foliant will do the following:

1. Download your `swagger.yaml` specification file over the link you supplied in config.
2. Pass `swagger.yaml` to the [Widdershins](#) tool which will convert it to Markdown.
3. Run some other specified preprocessors which may call [PlantUML](#) for generating diagrams or replace links to Jira issues with badges from [Shields.io](#), etc.
4. Glue all your separate md-source files into one with [Flatten](#) preprocessor.
5. Call [Slate](#) static site generator to build your site.

In this just one use case, Foliant had to internally run several third-party apps, each one of which needs to be installed on your computer before running the `foliant make site` command. It involves:

- Installing Widdershins, which is written on JS, so it will require [npm](#) and [Node.js](#) to be installed preliminarily.
- Installing PlantUML which requires [Java](#) and [Graphviz](#) installed preliminarily.
- Installing Slate, which requires [Ruby](#), [Bundler](#), and a bunch of other dependencies.

Woah! That's a lot of prerequisites! Now imagine you don't have any of them and you will have to spend half a day installing them and making sure they work. And now imagine that you want to build that static site on your home computer and you will need to waste another half a day just to do all that again.

That's where [Docker](#) steps in.

## How can Docker make your life easier

Docker allows you to run specific commands in an isolated, preconfigured environment, with no dependence on the system it is running on.

After reading the previous section, you now may have just one dream: how great would it be if I never needed to configure and install anything, and it would just work out of the box.

That was exactly our dream when Foliant started growing, so we've come up with a **Full Foliant Docker Image**, which has all the dependencies pre-installed. Not just backends and preprocessors, but all the open-source tools, which are used by them (except some proprietary tools like Oracle instant client).

It has [Pandoc](#) with [TexLive](#) for building beautiful and most complex PDFs, it has the most recent [Slate](#) static site generator with all the headache of installing Ruby dependencies already solved for you, it even has a [PostgreSQL](#) client if you ever need to generate docs for your database.

We keep the image up to date, and all new features that Foliant gets, shortly appear in there, so the only thing you should worry about is updating this image from time to time.

But there is a disk space issue. You see, since we've included almost everything you will ever need to use full Foliant's potential, the image had grown up to 2.5GB in compressed size and 6.5GB in unpacked form. That's mostly due to the TexLive package and some other bulky dependencies. All that will require decent Internet Bandwidth and some free disk space to download, which should not be that big of an issue in the modern world. If you prefer more compact installations, please, refer to the [Quickstart](#) where we show how to work with slimmer Foliant images.

But enough introductions, let's make it work!

## Getting Docker

The first step is to download and install Docker.

### Windows

Go to <https://www.docker.com/get-started> and download Docker installer.

Follow the instructions of the installer. In the end, it may ask you to restart the computer. After restarting, run Docker by the shortcut in your Start menu.

### Linux

Please, follow the [instructions](#) for your Linux distribution on the official docs.

After that [install Docker Compose](#).

### MacOs

Download and install Docker from [this page](#).

## Setting up Foliant project

Now that we've got Docker, we can create our test project.

Clone somewhere on your machine the [Foliant Project template](#). It's an empty Foliant project with the required file and directory structure. It also includes necessary Docker configs.

Open your terminal and `cd` to the directory with the cloned project template. You should see the following directory structure

```
1 $ tree
2 .
3   +-- Dockerfile
4   +-- README.md
5   +-- docker-compose.yml
6   +-- foliant.yml
7   +-- src
8     +-- index.md
```

Now let's build the Docker image:

```
1 $ docker-compose build
```

```
2 Building foliant
3 Step 1/1 : FROM foliant/foliant:full
4 full: Pulling from foliant/foliant
5 eeacba527962: Pull complete
6 25405ed4f245: Pull complete
7 ...
8 7ac153b46dbe: Pull complete
9 Digest: sha256:0
adac7993bb4f8178f62fc9bc1e8aa51a541629773a0f049b39949024e3b9353

10 Status: Downloaded newer image for foliant/foliant:full
11 ---> 51f465b95411load complete
12 Successfully built 51f465b95411
13 Successfully tagged test_foliant:latest
```

You will need to run this command every time after updating the image or editing Dockerfile. But you don't need to run it again if you edited any Foliant-related files, including source Markdown files or `foliant.yml`.

All preparations are done! Now let's build a site with MkDocs to test it out.

```
1 docker-compose run --rm foliant make site -w mkdocs
2 Creating network "test_default" with the default driver
3 Parsing config... Done
4 Applying preprocessor mkdocs... Done
5 Applying preprocessor _unescape... Done
6 Making site with MkDocs... Done—————
7
8 Result: New_Foliant_Project-2020-06-04.mkdocs
```

We used the `-w` option to tell Foliant that we want MkDocs backend to build our site. Full Foliant image contains many different backends, and several of them are capable of building `site` target. Without `-w` option Foliant will just ask you which backend to use.

Now open `index.html` inside the newly created `New_Foliant_Project-2020-06-04.mkdocs` folder, you should see something like this:



Documentation built with [MkDocs](#).

Documentation built with [MkDocs](#).

That means, it worked. Now you have all power of Foliant at your disposal!

One last thing: we keep developing Foliant and release new versions of different preprocessors almost every week, so make sure to update the Docker image every once in a while. To do this, run the command:

```
$ docker pull foliant/foliant:full
```

## Documenting API with Foliant

In this tutorial we will learn how to use Foliant to generate documentation from API specification formats [OpenAPI \(Swagger\)](#), [RAML](#) and [API Blueprint](#).

The general idea is that you supply a specification file path (`json` or `yaml` for OpenAPI, `raml` for RAML) to a preprocessor which will generate a Markdown document out of it. Markdown is what Foliant is good at, so after that you can do anything with it: convert to PDF, partially include in other documents, etc. In this guide we will concentrate on building a static website for your API documentation.

Please note that in this article we cover only the basic usage of the tools.

For detailed information on features and customizing output refer to each component's doc page.

# OpenAPI

## Installing prerequisites

Besides Foliant you will need to install some additional packages on your system. If you are using our full docker image `foliant/foliant:full`, you can skip this chapter.

First, install the [SwaggerDoc](#) preprocessor which will convert spec file to Markdown.

```
pip3 install foliantcontrib.swaggerdoc
```

SwaggerDoc preprocessor uses [Widdershins](#) under the hood, so you will need to install that too.

```
npm install -g widdershins
```

Finally, to build the static website we will be using [Slate backend](#):

```
pip3 install foliantcontrib.slate
```

Also note that Slate requires [Ruby](#) and [Bundler](#) to work (that's a lot of dependencies, I know).

## Creating project

Let's create Foliant project. The easiest way is to use `foliant init` command. After running the command Foliant will ask you about your project name. We've chosen "OpenAPI docs", but it may be anything:

```
1 cd ~/projects
2 foliant init
3     Enter the project name: OpenAPI docs
4     Generating project... Done
5
6 Project "OpenAPI docs" created in openapi-docs
```

In the output Foliant informs us that the project was created in a new folder `openapi-docs`. Let's copy your OpenAPI spec file into this folder:

```
cp ~/Downloads/my_api.yaml ~/projects/openapi-docs
```

In the end you should get the following directory structure:

```
1 └─
2   openapi-docs
3     ├── Dockerfile
4     ├── README.md
5     ├── docker-compose.yml
6     ├── foliant.yml
7     ├── my_api.yaml
8     ├── requirements.txt
9     └── src
10       └── index.md
```

If you wish to use Docker with full Foliant image, which is the recommended way to build Foliant projects, then open generated `Dockerfile` and replace its contents with the following line:

```
FROM foliant/foliant:full
```

## Configuring project

Now let's set up `foliant.yml`. Right now it looks like this:

```
1 title: OpenAPI docs
2
3 chapters:
4   - index.md
```

First add and fill up the  `preprocessors` section at the bottom:

```
1 preprocessors:
2   - swaggerdoc:
3     spec_path: !path my_api.yaml # path to your API spec
file, relative to project root
```

At this stage you may also specify path to custom templates dir in `environment : {user_templates: path/to/custom/templates}` parameter. Templates describe the exact way of how to convert structured specification file into a Markdown document. For this tutorial we will be using default templates because they are perfect for our static site. Check [Widdershins docs](#) for detailed info on templates.

The last thing we need to do is point Foliant where to insert the generated Markdown from the spec file. We already have a source file created for us by `init` command, called `index.md`, so let's use it to store our API docs.

Open `openapi-docs/src/index.md` with text editor and replace its contents with the following:

```
<swaggerdoc></swaggerdoc>
```

Foliant will insert generated markdown on the place of this tag during build. You may even add some kind of introduction for the API docs before the tag, if you don't have such inside your spec file.

That's it! All is left to do is run `make` command to build your site.

```
1 foliant make site --with slate
2     Parsing config... Done
3     Applying preprocessor swaggerdoc... Done
4     Applying preprocessor slate... Done
5     Applying preprocessor _unescape... Done
6     Making site...
7 ...
8     Done
9
10 Result: OpenAPI_docs-2019-11-29.slate/
```

If you use docker, the command is:

```
docker-compose run --rm foliant make site --with slate
```

Now if you open the `index.html` from just created `OpenAPI_docs-2019-11-29.slate` folder, you should see something like this:



**Figure 10.** Slate static site

You can customize the page styles, add or remove language example tabs and tune other options. Check the [Slate backend](#) documentation for details.

## RAML

Building API docs from RAML specification is quite similar to that of OpenAPI, the difference is that instead of `swaggerdoc` preprocessor you use `ramldoc`. We will go through all the steps anyway.

### Installing prerequisites

Besides Foliant you will need to install some additional packages on your system. If you are using our full docker image `foliant/foliant:full`, you can skip this chapter.

First, install the [RAMLdoc](#) preprocessor which will convert spec file to Markdown.

```
pip3 install foliantcontrib.ramldoc
```

RAMLdoc preprocessor uses `raml2html` with `full-markdown-theme` under the hood, so you will need to install those too.

```
npm install -g raml2html raml2html-full-markdown-theme
```

Finally, to build the static website we will be using [Slate backend](#). If you don't have it, run:

```
pip3 install foliantcontrib.slate
```

Also note that Slate requires [Ruby](#) and [Bundler](#) to work.

## Creating project

Let's create Foliant project. The easiest way is to use [foliant init](#) command. After running the command Foliant will ask you about your project name. We've chosen "API docs", but it may be anything:

```
1 cd ~/projects
2 foliant init
3     Enter the project name: API docs
4     Generating project... Done
5
6     Project "API docs" created in api-docs
```

In the output Foliant informs us that the project was created in a new folder `api-docs`. Now let's copy your RAML spec file to this folder:

```
cp ~/Downloads/my_api.raml ~/projects/api-docs
```

In the end you should get the following directory structure:

```
1 └──
2    api-docs
3     ├── Dockerfile
4     ├── README.md
5     ├── docker-compose.yml
6     ├── foliant.yml
7     ├── my_api.raml
8     ├── requirements.txt
9     └── src
10       └── index.md
```

If you wish to use Docker with full Foliant image, which is the recommended way to build Foliant projects, then open generated `Dockerfile` and replace its contents with the following line:

```
FROM foliant/foliant:full
```

## Configuring project

Now let's set up `foliant.yml`. Right now it looks like this:

```
1 title: API docs
2
3 chapters:
4   - index.md
```

First add and fill up the  `preprocessors` section at the bottom:

```
1 preprocessors:
2   - ramldoc:
3     spec_path: !path my_api.yaml # path to your API spec
file, relative to project root
```

At this stage you may also specify path to custom templates dir in the `template_dir` parameter. Templates describe the exact way of how to convert structured specification file into a Markdown document. `raml2html` uses [Nunjucks](#) templates, which are stored in the theme. So the easiest way to create your own templates is to copy [default](#) ones and adjust them to your needs. But we will use the default template which works great with Slate.

The last thing we need to do is point Foliant where to insert the generated Markdown from the spec file. We already have a source file created for us by `init` command, called `index.md`, so let's use it to store our API docs.

Open `api-docs/src/index.md` with text editor and replace its contents with the following:

```
<ramldoc></ramldoc>
```

Foliant will insert generated markdown on the place of this tag during build. You may even add some kind of introduction for the API docs before the tag, if you don't have such in your spec file.

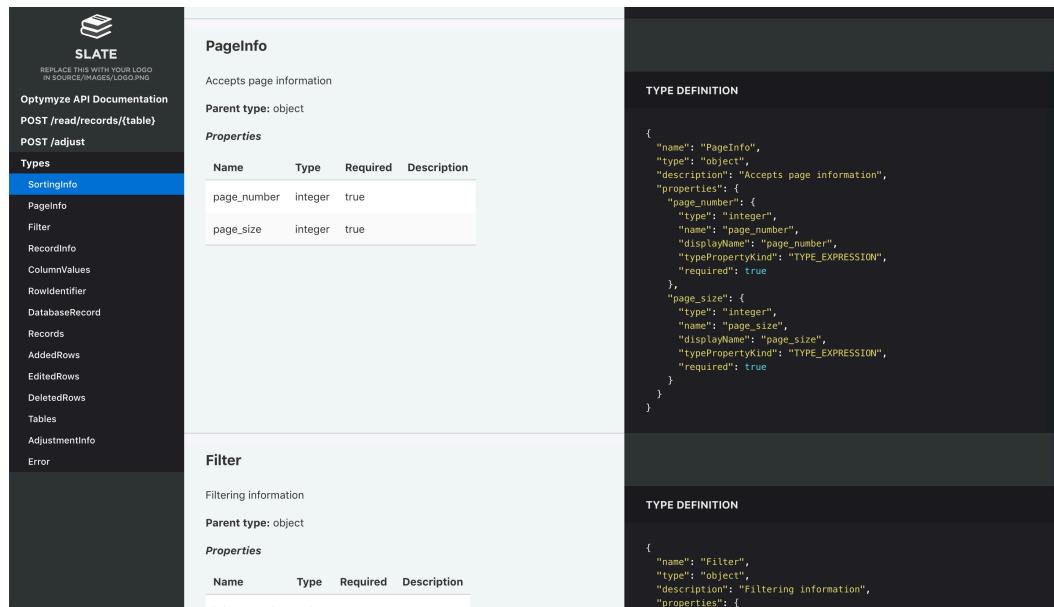
That's it! All is left to do is run `make` command to build your site.

```
1 foliant make site --with slate
2     Parsing config... Done
3     Applying preprocessor ramldoc... Done
4     Applying preprocessor slate... Done
5     Applying preprocessor _unescape... Done
6     Making site...
7 ...
8     Project built successfully.
9
10    Done
11
12    Result: API_docs-2019-11-29.slate/
```

If you use docker, the command is:

```
docker-compose run --rm foliant make site --with slate
```

Now if you open the `index.html` from just created `API_docs-2019-11-29.slate` folder, you should see something like this:



**Figure 11.** Slate static site

You can customize the page styles, add or remove language example tabs and tune other options. Check the [Slate backend](#) documentation for details.

## Blueprint

API Blueprint is a Markdown-based API specification format. That's why the build process differs from that of OpenAPI or RAML: we skip the converting step and just add the specification file as a source.

### Installing prerequisites

To build a static site we will use [Aglio](#) backend which is designed specifically for rendering API Blueprint. So first install the backend and [Aglio renderer](#) itself:

```
1 pip3 install foliantcontrib.aglio
2 npm install -g aglio
```

### Creating project

Let's create a Foliant project. The easiest way is to use [foliant init](#) command. After running the command Foliant will ask you about your project name. We've chosen "API docs", but it may be anything:

```
1 cd ~/projects
2 foliant init
3     Enter the project name: API docs
4     Generating project... Done
5
6     Project "API docs" created in api-docs
```

In the output Foliant informs us that the project was created in a new folder `api-docs`. Now copy your Blueprint spec file into the `src` subfolder (it's better to change the extension to `.md` too), replacing "index.md":

```
cp ~/Downloads/spec.abip ~/projects/api-docs/src/index.md
```

In the end you should get the following directory structure:

```
1 └──
2   └── openapi-docs
3     └── Dockerfile
```

```
4   └── README.md
5   └── docker-compose.yml
6   └── foliant.yml
7   └── requirements.txt
8   └── src
9     └── index.md
```

If you wish to use Docker with full Foliant image, which is the recommended way to build Foliant projects, then open generated `Dockerfile` and replace its contents with the following line:

```
FROM foliant/foliant:full
```

## Configuring project

Now check your `foliant.yml`. Right now it looks like this:

```
1 title: API docs
2
3 chapters:
4   - index.md # this should be your API Blueprint
  specification
```

It may be hard to believe, but no other configuration is required! Let's build our project:

```
1 foliant make site --with aglio
2     Parsing config... Done
3     Applying preprocessor flatten... Done
4     Applying preprocessor _unescape... Done
5     Making site... Done
6
7     Result: OpenAPI_docs-2019-11-29.aglio
```

If you use docker, the command is:

```
docker-compose run --rm foliant make site --with aglio
```

Now if you open the `index.html` from just created `API_docs-2019-11-29.aglio` folder, you should see something like this:



**Figure 12.** Aglio static site

It's not near as attractive as the Slate site we had in previous examples. But don't worry, Aglio supports styling with CSS and layout control with [Jade](#) templates. It also has several built-in themes, which look much better than the default one.

Open your `foliant.yml` again and add following lines at the end:

```

1 backend_config:
2     aglio:
3         params:
4             theme-variables: streak
5             theme-template: triple

```

Now run the same build command:

```
foliant make site --with aglio
```

And look at the result:



**Figure 13.** Aglio more beautiful static site

Much better!

## Documenting Databases with Foliant

### Introduction

In these tutorials we will show you a way to document your database using Foliant. Right now there are options available for those of you who use **PostgreSQL**, **Oracle** and **DBML**, but we will probably add more vendors later.

### The principles

Generally, we want to keep our docs as close to the code as possible. When documenting source code we usually can utilize the power of Swagger to generate docs from comments in the sources. Since there's no Swagger for databases, we had to invent something similar.

We are going to add actual descriptions of the tables and fields using comments. Comment (not to be confused with SQL script --comments) is a special entity which in one way or another is present in most DBMSs. They don't affect the data or table

structure, they are only used for documentation purposes. You can add a comment like this:

```
COMMENT ON TABLE "Clients" IS "Table holding info about the clients"
```

After describing all your entities inside your database, we need to get all this data somehow. For this, we will use Foliant [DBDoc preprocessor](#). It queries the database to get its structure (including our comments) and converts it into Markdown. We can then use this Markdown to generate a static site for our documentation.

## The tutorials

[Documenting with DBML specification](#)

[Documenting Oracle Database](#)

[Documenting PostgreSQL Database](#)

## Documenting DBML schema

Quote from the official website: [DBML \(Database Markup Language\)](#) is an open-source DSL language designed to define and document database schemas and structures. It is designed to be simple, consistent and highly-readable. And that makes it a perfect choice for designing your database. You can create your table structure without messing with cumbersome SQL in a more human-readable way like this:

```
1 Table users {
2     id integer
3     username varchar
4     role varchar
5     created_at timestamp
6 }
7
8 Table posts {
9     id integer [primary key]
10    title varchar
11    body text [note: 'Content of the post']
12    user_id integer
13    status post_status
14    created_at timestamp
```

```
15 }
16
17 Enum post_status {
18     draft
19     published
20     private [note: 'visible via URL only']
21 }
22
23 Ref: posts.user_id > users.id // many-to-one
```

As you may have noticed, DBML also has tools to document pieces of your schema using `notes` (`body text [note: 'Content of the post']`) and comments (`Ref: posts.user_id > users.id // many-to-one`).

So how can we convert DBML schema descriptions into a human-readable document? The idea is pretty simple: we parse the DBML definitions and pass them to a Jinja template, which renders markdown for us. After that we use one of our backends (we will use [Slate](#) in this tutorial) to build a static site out of it.

We won't need to do it all manually, of course, we just need to configure Foliant to do that for us.

## Installing prerequisites

If you are running Foliant natively, you will need to install some prerequisites. But if you are working with our [Full Foliant Docker image](#), you don't need to do that, just go to the next stage.

First, you will need Foliant, of course. If you don't have it yet, please, refer to the [installation guide](#).

Next, let's install [Foliant Init](#) to facilitate the task of creating new project:

```
$ pip3 install foliantcontrib.init
```

Install DBMLDoc and PlantUML preprocessors, and the Slate backend:

```
$ pip3 install foliantcontrib.dbmldoc foliantcontrib.slate,
foliantcontrib.plantuml
```

We are going to use [Slate](#) for building a static website with documentation, so you will need to [install Slate dependencies](#).

Finally, [install PlantUML](#), we will need it to draw database scheme.

## Creating project

Let's create a Foliant project for our experiments. `cd` into the directory where you want your project created and run the `init` command:

```
1 $ cd ~/foliant_projects
2 $ foliant init
3 Enter the project name: Database Docs
4 Generating project... Done—————
5
6 Project "Database Docs" created in database-docs
7
8 $ cd database-docs
```

The other option is to clone the [Foliant Project template](#) repository:

```
1 $ cd ~/foliant_projects
2 $ mkdir database-docs
3 $ git clone https://github.com/foliant-docs/
  foliant_project_template.git database-docs
4 Cloning into 'database-docs'...
5 remote: Enumerating objects: 11, done.
6 remote: Counting objects: 100% (11/11), done.
7 remote: Compressing objects: 100% (7/7), done.
8 remote: Total 11 (delta 1), reused 11 (delta 1), pack-reused
  0
9 Unpacking objects: 100% (11/11), done.
10 $ cd database-docs
```

Next, let's download the sample DBML spec and save it into file `schema.dbml` in the root your Foliant project:

```
$ wget https://raw.githubusercontent.com/holistics/dbml/
master/packages/dbml-core/_tests__parser/dbml-parse/input/
general_schema.in.dbml -O schema.dbml
```

## Setting up project

Now it's time to set up our config. Open `foliant.yml` and add the following lines:

```
1 title: Database Docs
2
3 chapters:
4   - index.md
5
6 + preprocessors:
7 +   - dbmldoc:
8 +     spec_path: schema.dbml
9 +   - plantuml
10 +
```

We've added the PlantUML and DBMLDoc preprocessors to the pipeline and specified path to our DBML sample schema. DBMLDoc will parse our schema and convert it into Markdown, plantuml will draw the visual diagram of our DB schema.

Note: if `plantuml` is not available under `$ plantuml` in your system, you will also need to specify path to `plantuml.jar` in preprocessor settings like this:

```
1   - plantuml:
2     plantuml_path: /usr/bin/plantuml.jar
```

Finally, we need to tell Foliant where in the source files should it insert the generated documentation. Since we already have an `index.md` chapter created for us by `init` command, let's put it in there. Open `src/index.md` and make it look like this:

```
1 # Welcome to Database Docs
2
3 -Your content goes here.
4 +<dbmldoc></dbmldoc>
5 +
```

## Building site

All preparations done, let's build our site:

```

1 $ foliant make site -w slate
2 Parsing config... Done
3 Applying preprocessor dbmldoc... Done
4 Applying preprocessor plantuml... Done
5 Applying preprocessor flatten... Done
6 Applying preprocessor _unescape... Done
7 Making site... Done
8 ...
9
10 Result: Database_Docs-2020-06-03.slate/

```

If you are using Docker, the command is:

```
$ docker-compose run --rm foliant make site -w slate
```

Now open `Database_Docs-2020-06-03.slate/index.html` and look what you've got:

column	properties	type	descr	fkey
<code>id</code>	<code>AUTOINCREMENT PRIMARY KEY</code>	<code>int</code>		
<code>user_id</code>	<code>NOT NULL UNIQUE</code>	<code>int</code>		
<code>status</code>			<code>orders_status</code>	
<code>created_at</code>				<code>varchar</code>

column	properties	type	descr	fkey
<code>order_id</code>		<code>int</code>	<code>orders.id</code>	
<code>product_id</code>		<code>int</code>	<code>products.id</code>	
<code>quantity</code>		<code>int</code>		

That looks good enough, but you may want to tweak the appearance of your site. You can edit the Jinja-template to change the way DBMLDoc generates markdown our of your schema. After first build, the default template should have appeared in your

project dir under the name `dbml.j2`. If you want to change the looks of your site, please, refer for instructions to the [Slate](#) backend documentation.

## Documenting Oracle Database

Please note that in this article we cover only the basic usage of the tools.

For detailed information on features and customizing output refer to each component's doc page.

### Installing prerequisites

First you will need to install some prerequisites. If you are running Foliant natively, follow the guide below. If you are working with our [Full Docker image](#), you don't need to do anything just now, you can skip to the next stage.

First, you will need Foliant, of course. If you don't have it yet, please, refer to the [installation guide](#).

Next, let's install [Foliant Init](#) to facilitate the task of creating new project:

```
$ pip3 install foliantcontrib.init
```

Install DBDoc and PlantUML preprocessors, and the Slate backend:

```
$ pip3 install foliantcontrib.dbdoc foliantcontrib.slate,  
foliantcontrib.plantuml
```

We are going to use [Slate](#) for building a static website with documentation, so you will need to [install Slate dependencies](#).

[Install PlantUML](#), we will need it to draw the database scheme.

Install [Oracle Instant Client](#), if you don't have it. We will need it to query the database.

### Creating project

Let's create a Foliant project for our experiments. `cd` to the directory where you want your project created and run the `init` command:

```
1 $ cd ~/foliant_projects  
2 $ foliant init  
3 Enter the project name: Database Docs  
4 Generating project... Done-----
```

```
5  
6 Project "Database Docs" created in database-docs  
7  
8 $ cd database-docs
```

The other option is to clone the [Foliant Project template](#) repository:

```
1 $ cd ~/foliant_projects  
2 $ mkdir database-docs  
3 $ git clone https://github.com/foliant-docs/  
    foliant_project_template.git database-docs  
4 Cloning into 'database-docs'...  
5 remote: Enumerating objects: 11, done.  
6 remote: Counting objects: 100% (11/11), done.  
7 remote: Compressing objects: 100% (7/7), done.  
8 remote: Total 11 (delta 1), reused 11 (delta 1), pack-reused  
    0  
9 Unpacking objects: 100% (11/11), done.  
10 $ cd database-docs
```

## Setting up project

Now it's time to set up our config. Open `foliant.yml` and add the following lines:

```
1 title: Database Docs  
2  
3 chapters:  
4   - index.md  
5  
6 + preprocessors:  
7 +   - dbdoc:  
8 +     host: localhost  
9 +     port: 1521  
10 +    dbname: orcl  
11 +    user: hr  
12 +    password: oracle  
13 +   - plantuml  
14 +
```

Make sure to use proper credentials for your Oracle database. If you are running Foliant from docker, you can use `host: host.docker.internal` to access `localhost` from docker.

Note: if plantuml is not available under `$ plantuml` in your system, you will also need to specify path to `plantuml.jar` in preprocessor settings like this:

```
1 - plantuml:  
2     plantuml_path: /usr/bin/plantuml.jar
```

Finally, we need to tell Foliant where in the source files should it insert the generated documentation. Since we already have an `index.md` chapter created for us by `init` command, let's put it in there. Open `src/index.md` and make it look like this:

```
1 # Welcome to Database Docs  
2  
3 -Your content goes here.  
4 +<dbdoc></dbdoc>  
5 +
```

If you are using Docker, you will also need to add Oracle Instant Client to your image. Since it is a proprietary software, we cannot include it in our Full Docker Image. But you can do it yourself. Our image is based on Ubuntu, so you can find instructions on how to install Oracle Instant Client on Ubuntu (spoiler: it's not that easy) and add those commands into the Dockerfile, or just find those commands made by someone else. For example, from this [Dockerfile by Sergey Makinen](#). Copy all commands starting from the third line into your `Dockerfile` and run `docker-compose build` to rebuild the image.

## Building site

All preparations done, let's build our site:

```
1 $ foliant make site -w slate  
2 Parsing config... Done  
3 Applying preprocessor dbdoc... Done  
4 Applying preprocessor plantuml... Done  
5 Applying preprocessor flatten... Done
```

```

6 Applying preprocessor _unescape... Done
7 Making site... Done
8 ...
9
10 Result: Database_Docs-2020-06-03.slate/

```

If you are using Docker, the command is:

```
$ docker-compose run --rm foliant make site -w slate
```

Now open `Database_Docs-2020-06-03.slate/index.html` and look what you've got:

The screenshot shows the SLATE database documentation interface. On the left, there's a sidebar with a logo, the word "SLATE", and a placeholder "REPLACE THIS WITH YOUR LOGO IN SOURCE/IMAGES/LOGO.PNG". Below this are sections for "Tables" (with links to COUNTRIES, DEPARTMENTS, EMPLOYEES, JOBS, JOB\_HISTORY, LOCATIONS, and REGIONS), "Triggers", and "Database Scheme". The main content area has a header "Tables" and a section titled "COUNTRIES". It contains a table with columns: column, nullable, type, descr, and fkey. The rows show: COUNTRY\_ID (N, CHAR, Primary key of countries table), COUNTRY\_NAME (Y, VARCHAR2, Country name), and REGION\_ID (Y, NUMBER, Region ID for the country. Foreign key to region\_id column in the departments table). Below this is a section titled "DEPARTMENTS" with a table showing: DEPARTMENT\_ID (N, NUMBER, Primary key column of departments).

column	nullable	type	descr	fkey
COUNTRY_ID	N	CHAR	Primary key of countries table.	
COUNTRY_NAME	Y	VARCHAR2	Country name	
REGION_ID	Y	NUMBER	Region ID for the country. Foreign key to region_id column in the departments table.	REGIONS

column	nullable	type	descr	fkey
DEPARTMENT_ID	N	NUMBER	Primary key column of departments	

That looks good enough, but you may want to tweak the appearance of your site. You can edit the Jinja-template to change the way DBMLDoc generates markdown our of your schema. The default template can be found [here](#). Edit it and save in your project dir, then specify in the `doc_template` parameter. If you want to change the looks of you site, please, refer for instructions to the [Slate](#) backend documentation.

# Documenting PostgreSQL Database

Please note that in this article we cover only the basic usage of the tools. For detailed information on features and customizing output refer to each component's doc page.

## Installing prerequisites

First you will need to install some prerequisites. If you are running Foliant natively, follow the guide below. If you are working with our [Full Docker image](#), you don't need to do anything, you can skip to the next stage.

First, you will need Foliant, of course. If you don't have it yet, please, refer to the [installation guide](#).

Next, let's install [Foliant Init](#) to facilitate the task of creating new project:

```
$ pip3 install foliantcontrib.init
```

Install DBDoc and PlantUML preprocessors, and the Slate backend:

```
$ pip3 install foliantcontrib.dbdoc foliantcontrib.slate,  
foliantcontrib.plantuml
```

We are going to use [Slate](#) for building a static website with documentation, so you will need to [install Slate dependencies](#).

[Install PlantUML](#), we will need it to draw the database scheme.

## Creating project

Let's create a Foliant project for our experiments. `cd` to the directory where you want your project created and run the `init` command:

```
1 $ cd ~/foliant_projects  
2 $ foliant init  
3 Enter the project name: Database Docs  
4 Generating project... Done—————  
5  
6 Project "Database Docs" created in database-docs  
7  
8 $ cd database-docs
```

The other option is to clone the [Foliant Project template](#) repository:

```
1 $ cd ~/foliant_projects
2 $ mkdir database-docs
3 $ git clone https://github.com/foliant-docs/
  foliant_project_template.git database-docs
4 Cloning into 'database-docs'...
5 remote: Enumerating objects: 11, done.
6 remote: Counting objects: 100% (11/11), done.
7 remote: Compressing objects: 100% (7/7), done.
8 remote: Total 11 (delta 1), reused 11 (delta 1), pack-reused
  0
9 Unpacking objects: 100% (11/11), done.
10 $ cd database-docs
```

## Setting up project

Now it's time to set up our config. Open `foliant.yml` and add the following lines:

```
1 title: Database Docs
2
3 chapters:
4   - index.md
5
6 + preprocessors:
7 +   - dbdoc:
8 +     host: localhost
9 +     port: 5432
10 +    dbname: postgres
11 +    user: postgres
12 +    password: postgres
13 +   - plantuml
14 +
```

Make sure to use proper credentials for your PostgreSQL database. If you are running Foliant from docker, you can use `host: host.docker.internal` to access `localhost` from docker.

Note: if plantuml is not available under `$ plantuml` in your system, you will also need to specify path to plantuml.jar in preprocessor settings like this:

```
1 - plantuml:  
2     plantuml_path: /usr/bin/plantuml.jar
```

Finally, we need to tell Foliant where in the source files should it insert the generated documentation. Since we already have an `index.md` chapter created for us by `init` command, let's put it in there. Open `src/index.md` and make it look like this:

```
1 # Welcome to Database Docs  
2  
3 -Your content goes here.  
4 +<dbdoc></dbdoc>  
5 +
```

If you are using Docker, you will also need to add Oracle Instant Client to your image. Since it is a proprietary software, we cannot include it in our Full Docker Image. But you can do it yourself. Our image is based on Ubuntu, so you can find instructions on how to install Oracle Instant Client on Ubuntu (spoiler: it's not that easy) and add those commands into the Dockerfile, or just find those commands made by someone else. For example, from this [Dockerfile by Sergey Makinen](#). Copy all commands starting from the third line into your `Dockerfile` and run `docker-compose build` to rebuild the image.

## Building site

All preparations done, let's build our site:

```
1 $ foliant make site -w slate  
2 Parsing config... Done  
3 Applying preprocessor dbdoc... Done  
4 Applying preprocessor plantuml... Done  
5 Applying preprocessor flatten... Done  
6 Applying preprocessor _unescape... Done  
7 Making site... Done  
8 ...  
9
```

<sup>10</sup> Result: Database\_Docs-2020-06-03.slate/

If you are using Docker, the command is:

```
$ docker-compose run --rm foliant make site -w slate
```

Now open Database\_Docs-2020-06-03.slate/index.html and look what you've got:

The screenshot shows the Foliant Slate backend interface. On the left, there's a sidebar with a logo, the word "SLATE", and a placeholder "REPLACE THIS WITH YOUR LOGO IN SOURCE/IMAGES/LOGO.PNG". Below this are sections for "Tables", "Triggers", and "Database Scheme". Under "Tables", there are links for COUNTRIES, DEPARTMENTS, EMPLOYEES, JOBS, JOB\_HISTORY, LOCATIONS, and REGIONS. The main content area has a title "Tables" and a section titled "COUNTRIES". It contains a table with columns: column, nullable, type, descr, and fkey. The first row (COUNTRY\_ID) is highlighted. The second row (COUNTRY\_NAME) is also highlighted. The third row (REGION\_ID) is shown with a note about being a foreign key to the REGIONS table. Below this is a section titled "DEPARTMENTS" with a table showing a single row for DEPARTMENT\_ID.

column	nullable	type	descr	fkey
COUNTRY_ID	N	CHAR	Primary key of countries table.	
COUNTRY_NAME	Y	VARCHAR2	Country name	
REGION_ID	Y	NUMBER	Region ID for the country. Foreign key to region_id column in the departments table.	REGIONS

column	nullable	type	descr	fkey
DEPARTMENT_ID	N	NUMBER	Primary key column of departments	

That looks good enough, but you may want to tweak the appearance of your site. You can edit the Jinja-template to change the way DBMLDoc generates markdown our of your schema. The default template can be found [here](#). Edit it and save in your project dir, then specify in the `doc_template` parameter. If you want to change the looks of you site, please, refer for instructions to the [Slate](#) backend documentation.

## Developer's Reference

You're welcome to contribute to Foliant and its extensions. Foliant ecosystem, of course, needs a lot of new useful extensions for all occasions.

Foliant and its extensions are developed in Git repositories that belong to the [foliant-docs](#) GitHub group.

The repo of Foliant Core is called [foliant](#). Repositories of Foliant extensions have names starting with the `foliantcontrib.` prefix. The repo of this documentation is called [docs](#).

Foliant Core modules and base classes that should be used in all newly developed extensions, are described at this page.

## Core Modules

Foliant Core includes a number of modules:

- `foliant`:
- `backends`:
  - `base`—defines the base class for all backends, see below;
  - `pre`—simplest backend that returns Markdown content processed by specified preprocessors as build result;
- `preprocessors`:
  - `base`—defines the base class for all preprocessors, see below;
  - `_unescape`—simple preprocessor that allows to use pseudo-XML tags that are recognized by other preprocessors as control sequences, 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` removes such characters. Instead of the `_unescape` preprocessor, it's recommended to use more flexible [EscapeCode](#) and [UnescapeCode](#) preprocessors;
- `cli`—defines the Foliant's root class `Foliant()` and the `entry_point()` method that is used as a starting point when calling Foliant; nested modules:
  - `base`—defines the base class for all CLI extensions, see below;
  - `make`—provides the main Foliant's command `make`;
- `config`:
  - `base`—defines the base class for all config extensions;
  - `include`—resolves the `!include` YAML tag that allows to reuse the data of one YAML file in another;
  - `path`—resolves the `!path` YAML tag. The string preceded by this modifier should be converted into an existing path relative to the Foliant project's top-level (“root”) directory;
  - `utils`—defines some basic methods that may be used in extensions of different types.

## The make() Method Arguments

To build any Foliant project, the method `make()` that is defined in the module `foliant.cli.make` should be called in one or another way.

The method takes a number of arguments; some of them pass onwards to backends and preprocessors:

- `target` (string)—required resulting target of the current build;
- `backend` (string, defaults to an empty string)—backend that is used for the current build;
- `project_path` (path, defaults to the current directory path)—path of top-level, “root” directory of the current Foliant project;
- `config_file_name` (string, defaults to `foliant.yml`)—Foliant project’s config file name;
- `quiet` (boolean, default to `False`)—flag that prohibits writing to `STDOUT`;
- `keep_tmp` (boolean, defaults to `False`)—flag that tells Foliant and its extensions not to delete a temporary working directory that is used during build;
- `debug` (boolean, defaults to `False`)—flag that tells Foliant and its extensions to log events of `info` and `debug` levels in addition to messages of `warning`, `error`, and `critical` levels.

## Base Classes

Foliant Core provides 4 base classes—one per each type of extensions.

- `BaseBackend()` that is defined in the `foliant.backends.base` module, is the base class for all backends. Each newly developed backend should:
  - be a module `foliant.backends.<your_backend_name>`;
  - import the class `BaseBackend()` from the `foliant.backends.base` module;
  - define its own class called `Backend()` that is inherited from `BaseBackend()`;
  - define the method called `make()` within the class `Backend()`.
- `BasePreprocessor()` that is defined in the `foliant.preprocessors.base` module, is the base class for all preprocessors. Each newly developed preprocessor should:
  - be a module `foliant.preprocessors.<your_preprocessor_name>`;
  - import the class `BasePreprocessor()` from the `foliant.preprocessors.base` module;

- define its own class called `Preprocessor()` that is inherited from `BasePreprocessor()`;
- define the method called `apply()` within the class `Preprocessor()`.
- `BaseCli()` that is defined in the `foliant.cli.base` module, is the base class for all CLI extensions. Each newly developed CLI extension should:
  - be a module `foliant.cli.<your_cli_extension_name>`;
  - import the class `BaseCli()` from the `foliant.cli.base` module;
  - define its own class called `Cli()` that is inherited from `BaseCli()`.
- `BaseParser()` that is defined in the `foliant.config.base` module, is the base class for all config extensions. Each newly developed config extension should:
  - be a module `foliant.config.<your_config_extension_name>`;
  - import the class `BaseParser()` from the `foliant.config.base` module;
  - define its own class called `Parser()` that is inherited from `BaseParser()`.

Quick reference on important variables that are provided by base classes, is below.

## The `BaseBackend()` Variables

- Class variables:
  - `targets` (tuple of strings) – names of targets that the backend can build;
  - `required_preprocessors_before` (tuple of strings) – names of preprocessors that should be applied before all other preprocessors when this target is used;
  - `required_preprocessors_after` (tuple of strings) – names of preprocessors that should be applied after all other preprocessors when this target is used;
- instance variables:
  - `self.context` – dictionary that contains the keys:
    - `project_path` (path) – path to the currently built Foliant project;
    - `config` (dictionary) – currently built Foliant project's full config;
    - `target` (string) – name of the resulting target;
    - `backend` (string) – name of the backend that is used in the current build;
    - `self.config` – exactly the same as `self.context['config']`;
    - `self.project_path` – exactly the same as `self.context['project_path']`;
    - `self.working_dir` (path) – exactly the same as `self.project_path / self.config['tmp_dir']`, the path to the temporary working directory that is used during build;
    - `self.logger` – logger instance;

- `self.quiet` (boolean) – if `True`, the backend should not write anything to `STDOUT`;
- `self.debug` (boolean) – if `True`, the backend should log the messages of `info` and `debug` levels.

## The `BasePreprocessor()` Variables

- Class variables:
  - `defaults` (dictionary) – default values of options that may be overridden in config;
  - `tags` (tuple of strings) – names of pseudo-XML tags that are recognized by the preprocessor, without `<` and `>` characters;
- instance variables:
  - `self.context` – dictionary that contains the keys:
    - `project_path` (path) – path to the currently built Foliant project;
    - `config` (dictionary) – currently built Foliant project's full config;
    - `target` (string) – name of the resulting target;
    - `backend` (string) – name of the backend that is used in the current build;
  - `self.config` – exactly the same as `self.context['config']`;
  - `self.project_path` – exactly the same as `self.context['project_path']`;
  - `self.working_dir` (path) – exactly the same as `self.project_path / self.config['tmp_dir']`, the path to the temporary working directory that is used during build;
  - `self.logger` – logger instance;
  - `self.quiet` (boolean) – if `True`, the preprocessor should not write anything to `STDOUT`;
  - `self.debug` (boolean) – if `True`, the preprocessor should log the messages of `info` and `debug` levels.
  - `self.options` (dictionary) – preprocessor's options, i.e. `{**self.defaults, **options}` where `options` is data that is read from config;
  - `self.pattern` – regular expression `pattern` that is used to get components of a pseudo-XML tag in easy way. Defined if `self.tags` is not empty. Provides the groups with the following names:
    - `tag` – for tag name;
    - `options` – for tag attributes (options) as a string; this string may be converted into a dictionary by using the `self.get_options()` method provided by the basic class;

- `body`—for tag body, i.e. a content between the opening and closing tags.

## The `BaseCli()` Variables

- Instance variables:
  - `self.logger`—logger instance.

## The `BaseConfig()` Variables

- Instance variables:
  - `self.project_path` (path)—path to the currently built Foliant project;
  - `self.config_path` (path)—path to the currently built Foliant project's config file;
  - `self.logger`—logger instance;
  - `self.quiet` (boolean)—if `True`, the config extension should not write anything to `STDOUT`.

# Metadata

This extension adds metadata support for Foliant. It also allows to add meta commands which use project's metadata and are called like this: `foliant meta <command>`. Finally, it adds the `meta generate` command to Foliant, which generates the yaml-file with project metadata.

## Installation

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

## Specifying metadata

Metadata for the [main section](#) (more on sections in [User's Guide](#) below) may be specified in the beginning of a Markdown-file using [YAML Front Matter](#) format:

```
1 ---
2 id: MAIN_DOC
3 title: Description of the product
4 key: value
5 ---
```

You may also use regular XML-like format with `meta` tag:

```
1 <meta
2     id="MAIN_DOC"
3     title="Description of the product"
4     key="value">
5 </meta>
```

If `meta` tag is present, all Metadata from YAML Front Matter is ignored.

## User's guide

Metadata allows you to specific properties to your documents, which won't be visible directly to the end-user. These properties may be:

- the document author's name;
- Jira ticket id;

- date of last revision;
- or anything else, there is no limitation.

This module is required for metadata to work in your projects. But it doesn't care about most of the fields and their values. The only exception being the `id` field. See **Special fields** section.

## Sections

You can specify metadata for a whole chapter and for its portions, which are called sections. Section is a fragment of the document from one heading to another one of the same level of higher.

Metadata specified at the beginning of the document (before the first heading) is applied to the whole Markdown document. We call it the main section of the chapter.

Note that you can specify metadata for the main section either in YAML Front Matter format, or with `meta` tag.

If you specify metadata after the heading of some level, it will be applied to all content inside this heading, including the nested headings. See the illustration below.

```

---  

id: main  

author: John Smith  

---  

# Introduction  

<meta id="intro" type="optional"></meta>  


  Lorem ipsum dolor sit amet, consectetur adipisicing elit. Sed sit ea ducimus nemo, aut. Odio aspernatur ut laudantium nihil nam. Temporibus explicabo, porro consectetur quia.

## Requirements  

<meta id="recs" type="optional"></meta>  


  Lorem ipsum dolor sit amet, consectetur adipisicing elit. Eligendi modi vero deserunt, itaque odio quo!

#### Recommendations  


  Lorem ipsum dolor sit amet, consectetur adipisicing elit. Fuga, rem.


  Cum fugit omnis officia quia reprehenderit voluptate perspiciatis iusto iste, magni magnam.

## Description  

<meta id="descr" hotel="trivago"></meta>  


  Lorem ipsum dolor sit amet, consectetur adipisicing elit. Aperiam, fugit.


  Culpa fuga autem nihil molestiae tenetur, dolorem recusandae officiis iusto, pariatur voluptatem illo distinctio officia quae, excepturi error ducimus minus temporibus laborum maxime qui quam? Obcaecati, facilis.

# Summary  


  Tenetur magnam enim est nihil nemo iusto repudiandae corrupti officiis dignissimos error. Atque tempore minus placeat aliquid nulla similiique repudiandae non corporis omnis eum doloribus magnam, officia temporibus.


```

## Special fields

Right now there's only one field that is treated specially: the `id` field.

If specified, it will be used as identifier of the section. Note that IDs must be unique within the project.

If `id` field is omitted – the section will get auto generated id based on:

- chapter filename for main section,
- title for general sections.

## Additional info

Metadata works only for files, mentioned in the `chapters` section in `foliant.yml`. All other files in `src` dir are ignored.

When using `includes`, all metadata from the included content is removed.

# Developer's guide

You can use the powers of metadata in your preprocessors, backends and other tools. You can define fields with special meaning for your tools and process sections based on these fields.

## Getting metadata

Typical way to work with metadata is to run the `load_meta` function from the `foliant.meta.generate` module.

**`load_meta(chapters: list, md_root: str or PosixPath = 'src') -> Meta`**

This function collects metadata and returns a `Meta` object, which gives access to all sections and meta-fields in the project.

The required parameter is `chapters` – list of chapters loaded from `foliant.yml`

```
1 >>> from foliant.meta.generate import load_meta
2 >>> meta = load_meta(['index.md'])
```

You can also specify the `md_root` parameter. If your tool is a CLI extension, `md_root` should point to the project's `src` dir. But if you are building a preprocessor or a backend, you would probably want to point it to the `__folianttmp__` dir with the current state of the sources.

## The Meta class

Meta class holds all metadata and offers few handy methods to work with it.

**`load_meta_from_file(filename: str or PosixPath)`**

This method allows you to load meta into the `Meta` class instance from previously generated yaml-file. Use it only with empty `Meta` class:

```
1 >>> from foliant.meta.classes import Meta
2 >>> meta = Meta()
```

```
3 >>> meta.load_meta_from_file('meta.yml')
```

### **iter\_sections()**

This method returns an iterator which yields project's meta-sections ( `Section` objects) in the proper order from the first chapter to the last one.

### **get\_chapter(self, filename: str or PosixPath) -> Chapter**

Get chapter ( `Chapter` object) by its path. `filename` should be path to chapter relative to the Project dir (or absolute path).

### **get\_by\_id(self, id\_: str) -> Section**

Get section ( `Section` object) by its id.

### **chapters**

A property which holds the list of chapters ( `Chapter` objects).

## The Chapter class

`Chapter` class represents a project's chapter. It has several important methods which may be useful for working with metadata.

### **iter\_sections()**

This method returns an iterator which yields chapter's meta-sections ( `Section` objects) in the proper order from the first chapter to the last one.

### **get\_section\_by\_offset(offset: int) -> Section:**

This method allows you to get section ( `Section` object) by just pointing to a place in text. Pointing is performed by specifying offset from the beginning of the file in `offset` parameter.

### important properties

#### **main\_section**

A property which holds the main section of the chapter.

#### **name**

Chapter's name as stated in foliant.yml.

#### **filename**

Chapter's filename.

## The Section class

Section represents a meta section.

### **iter\_children()**

This method returns an iterator which yields the section's child sections (Section objects) in the proper order.

### **get\_source(self, without\_meta=True) -> str**

Returns section's source. The section title is also included in the output. If `without_meta` is True, all meta tags are cut out from the text.

### **is\_main(self) -> bool**

Determine whether the section is main or not.

### important properties

#### **id**

Holds section's ID.

#### **title**

Section's title.

#### **chapter**

Holds reference to section's chapter.

#### **children**

Holds list of section's children in proper order.

#### **data**

Holds a dictionary with fields and their values, defined in the `<meta>` tag (or YAML front matter if it is a main section).

#### **level**

Section's level. Main section has level 0, section, defined inside the `###` heading will have level 3.

#### **start and end**

Section's offsets from the beginning of the chapter.

#### **filename**

Holds reference to section's chapter's filename for easy access.

# Backends

## Aglio

pypi v1.0.0

GitHub v1.0.0

## Aglio Backend for Foliant

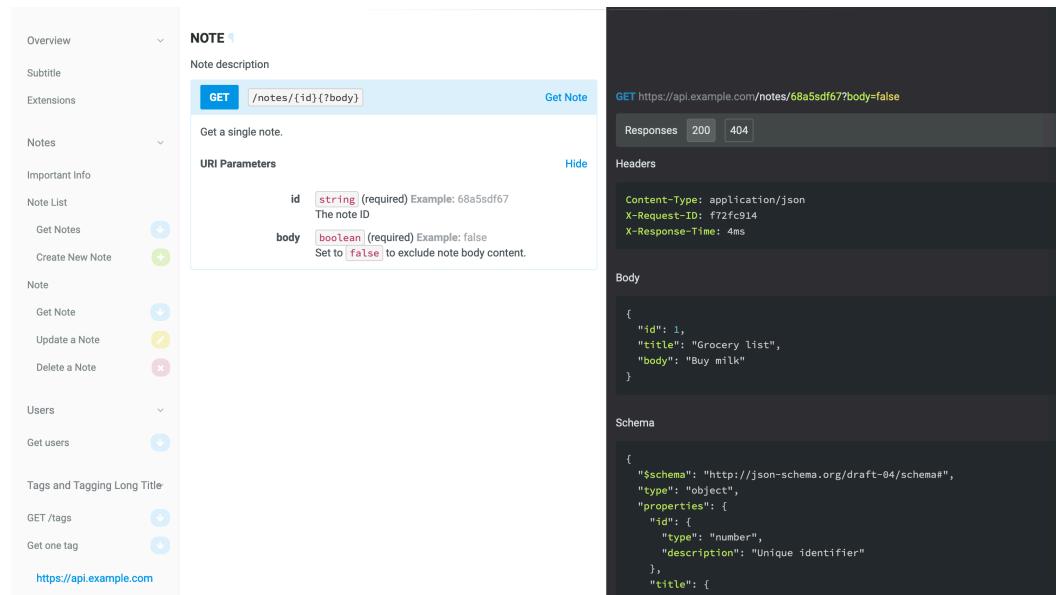


Figure 14. Static site built with Aglio backend

Aglio backend generates API documentation from [API Blueprint](#) using [aglio renderer](#). This backend operates the `site` target.

Note, that aglio is designed to render API Blueprint documents. Blueprint syntax is very close to that of Markdown and you may be tempted to use

this backend as a general purpose static site generator. It may work in some cases, but is not guaranteed to work in all of them.

## Installation

```
$ pip install foliantcontrib.aglio
```

To use this backend [Aglio should be installed](#) on your system:

```
$ npm install -g aglio
```

To test if you've installed aglio properly run the `aglio -h` command, which should return you a list of options.

## Usage

To generate a static website from your Foliant project run the following command:

```
1 $ foliant make site --with aglio
2 Parsing config... Done
3 Applying preprocessor flatten... Done
4 Applying preprocessor _unescape... Done
5 Making site... Done
6
7 Result: My_Awesome_Project.aglio
```

## Config

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

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

```
1 backend_config:
2   aglio:
3     aglio_path: aglio
4     params:
5       theme-variables: flatly
6       fullWidth: True
```

**aglio\_path** Path to the aglio binary. Default: `aglio`

**params** Parameters which will be supplied to the `aglio` command. To get the list of possible parameters, run `aglio -h` or check the [official docs](#).

## Customizing output

### Templates

You can customize the appearance of the static website build by aglio with [Jade](#) templates. Aglio has two built-in templates:

- `default` – two-column web-page;
- `triple` – three-column web-page.

To add your own template, follow [the instructions](#) in the official docs.

To specify the template add the `theme-template` field to the `params` option:

```
1 backend_config:  
2   aglio:  
3     params:  
4       theme-template: triple
```

### Color scheme

You can customize the color scheme of the website by specifying the color scheme name in `theme-variables` param.

Available built-in color schemes:

- `default`,
- `cyborg`,
- `flatly`,
- `slate`,
- `streak`.

You can also specify your own scheme in a LESS or CSS file.

```
1 backend_config:  
2   aglio:  
3     params:  
4       theme-variables: flatly
```

## Stylesheets

Finally, you can provide custom stylesheets in a LESS or CSS file in `theme-style` param:

```
1 backend_config:  
2   aglio:  
3     params:  
4       theme-style: !path my-style.less
```

## Confluence

pypi v0.6.12

GitHub v0.6.12

The screenshot shows a Confluence page with the following components:

- Left Sidebar:** Contains a tree view of the page structure, including sections like "Empty Page", "Errors", "New article", etc.
- Main Content Area:**
  - mermaid diagram:** A flowchart illustrating a process involving events X and Y and a transition to state Z.
  - Code Block:** Displays Python code for a class named `QueryBase` with methods `__init__` and `_resolve_filters`.
  - Table:** A table with columns "Tables", "Are", and "Cool" containing four rows of data: col1, col2, col3.
- Bottom Navigation:** Includes "Space tools" and a back arrow.

**Figure 15.** Confluence page built with 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 on your system. Pandoc is needed to convert Markdown into HTML.

## Usage

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

```
1 $ foliant make confluence
2 Parsing config... Done
3 Making confluence... Done—————
4
5 Result:
6 https://my_confluence_server.org/pages/viewpage.action?
  pageId=123 (Page Title)
```

## Config

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

Specify all options in `backend_config.confluence` section:

```
1 backend_config:
2   confluence:
3     passfile: confluence_secrets.yml
4     host: 'https://my_confluence_server.org'
5     login: user
6     password: user_password
7     id: 124443
8     title: Title of the page
```

```

9   space_key: "~user"
10  parent_id: 124442
11  parent_title: Parent
12  test_run: false
13  notify_watchers: false
14  toc: false
15  nohead: false
16  restore_comments: true
17  resolve_if_changed: false
18  pandoc_path: pandoc
19  codeblocks:
20  ...

```

**passfile** Path to YAML-file holding credentials. See details in [Supplying Credentials](#) section. Default: `confluence_secrets.yml`

**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 the build.

**password** Password of the user. If the password is not supplied, it will be prompted during the 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 one space must 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](#).

**parent\_title** Another way to define the parent of the page. Lower priority than parent\_id. Title of the parent page under which the new one(s) should be created. The parent should exist under the space\_key specified. [Only for not yet existing pages](#).

**test\_run** If this option is true, Foliant will prepare the files for uploading to Confluence, but won't actually upload them. Use this option for testing your content before upload. The prepared files can be found in `.confluencecache/debug` folder. Default: `false`

**notify\_watchers** If `true` – watchers will be notified that the page has changed.  
Default: `false`

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

**nohead** If set to `true`, first title will be removed from the page. Use it if you are experiencing duplicate titles. 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 the comment as resolved. Default: `false`

**pandoc\_path** Path to Pandoc executable (Pandoc is used to convert Markdown into HTML).

**codeblocks** Configuration for converting Markdown code blocks into code-block macros. See details in **Code blocks processing** sections.

## 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:

```
1 ---
2 field: value
3 field2: value
4 ---
5
6 Your chapter md-content
```

or like this:

```
1 <meta
2     field="value"
3     field2="value">
```

```
4 </meta>
5
6 Your chapter md-content
```

The result of the above examples will be exactly the same. Just remember that first syntax, with three dashes – will only work if it is in the beginning of the document. For all other cases use the meta-tag syntax.

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

```
1 ---
2 confluence:
3   title: My confluence page
4   space_key: "~user"
5 ---
6
7 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 the page with `id2`.

**Notice** You can omit `title` param in metadata. In this case section heading will be used as a title.

If you want to upload just a part of the chapter, specify meta tag under the heading, which you want to upload, like this:

```
1 # My document
2
3   Lorem ipsum dolor sit amet, consectetur adipisicing elit.
4     Explicabo quod omnis ipsam necessitatibus, enim voluptatibus
5     .
6
7 ## Components
8
```

```

7 <meta
8     confluence=""
9         title='System components'
10        space_key='~user'
11    ">
12 </meta>
13
14 Lorem ipsum dolor sit amet, consectetur adipisicing elit.
15 Vel, atque!
16 ...

```

In this example, only the **Components** section with all its content will be uploaded to Confluence. The **My document** heading will be ignored.

### 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 that 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:

```

1 backend_config:
2     confluence:
3         host: https://my_confluence_server.org
4         login: user
5         password: pass
6         title: My unique title
7         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 the previous section). If the page with a 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.

```
1 backend_config:  
2   confluence:  
3     host: https://my_confluence_server.org  
4     login: user  
5     password: pass  
6     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:

```
1 backend_config:  
2   confluence:  
3     host: https://my_confluence_server.org  
4     login: user  
5     password: pass  
6     id: 124443  
7     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.

**Known issue:** right now this mode doesn't work with layout sections. If you are using sections, whole content will be overwritten.

### 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:

```
1 Lorem ipsum dolor sit amet, consectetur adipisicing elit.  
2   Odit dolorem nulla quam doloribus delectus voluptate.  
3  
4 <raw_confluence><ac:structured-macro ac:macro-id="1" ac:name  
5   ="toc" ac:schema-version="1"/></raw_confluence>  
6  
7 Lorem ipsum dolor sit amet, consectetur adipisicing elit.  
8   Officiis, laboriosam cumque soluta sequi blanditiis,  
9   voluptatibus quaerat similique nihil debitis repellendus.
```

### Code blocks processing

Since 0.6.9 backend converts Markdown code blocks into Confluence code-block macros. You can tune the macros appearance by specifying some options in `codeblocks` config section of Confluence backend

```
1 backend_config:  
2   confluence:  
3     codeblocks: # all are optional  
4       theme: django  
5       title: Code example  
6       linenumbers: false  
7       collapse: false
```

**theme** Color theme of the code blocks. Should be one of:

- emacs,
- django,

- fadetogrey,
- midnight,
- rdark,
- eclipse,
- confluence.

**title** Title of the code block.

**linenumbers** Show line numbers in code blocks. Default: `false`

**collapse** Collapse code blocks into a clickable bar. Default: `false`

Right now Foliant only converts code blocks by backticks/tildes (tabbed code blocks are ignored for now):

1 This code block will be converted:

```

2
3 `` `python
4 def test2():
5     pass
6 `` ``
```

1 And this:

```

2 ~~~
3 def test3():
4     pass
5 ~~~
```

Syntax name, defined after backticks/tildes is converted into its Confluence counterpart. Right now following syntaxes are supported:

- actionscript,
- applescript,
- bash,
- c,
- c,
- coldfusion,
- cpp,
- cs,
- css,
- delphi,

- diff,
- erlang,
- groovy,
- html,
- java,
- javascript,
- js,
- perl,
- php,
- powershell,
- python,
- xml,
- yaml.

## Supplying Credentials

There are two ways to supply credentials for your confluence server.

### 1. In foliant.yml

The most basic way is just to put credentials in foliant.yml:

```
1 backend_config:  
2     confluence:  
3         host: https://my_confluence_server.org  
4         login: user  
5         password: pass
```

It's not very secure because foliant.yml is usually visible to everybody in your project's git repository.

### 2. Using passfile

Alternatively, you can use a passfile. Passfile is a yaml-file which holds all your passwords. You can keep it out from git-repository by storing it only on your local machine and production server.

To use passfile, add a `passfile` option to foliant.yml:

```
1 backend_config:  
2     confluence:  
3         host: https://my_confluence_server.org
```

```
4      passfile: confluence_secrets.yaml
```

The syntax of the passfile is the following:

```
1 hostname:  
2     login: password
```

For example:

```
1 https://my_confluence_server.org:  
2     user1: wFwG34uK  
3     user2: MEUeU3b4  
4 https://another_confluence_server.org:  
5     admin: adminpass
```

If there are several records for a specified host in passfile (like in the example above), Foliant will pick the first one. If you want specific one of them, add the login parameter to your foliant.yml:

```
1 backend_config:  
2     confluence:  
3         host: https://my_confluence_server.org  
4         passfile: confluence_secrets.yaml  
5         login: user2
```

## Credits

The following wonderful tools and libraries are used in foliantcontrib.confluence:

- [Atlassian Python API wrapper](#),
- [BeautifulSoup](#),
- [PyParsing](#),
- [Pandoc](#).

## MdToPdf

[pypi](#) v1.0.0

## MdToPdf backend for Foliant

This backend generates a single PDF document from your Foliant project. It uses [md-to-pdf](#) library under the hood.

md-to-pdf supports styling with CSS, automatic syntax highlighting by [highlight.js](#), and PDF generation with [Puppeteer](#).

MdToPdf backend for Foliant operates the `pdf` target.

### Installation

First install md-to-pdf on your machine:

```
$ npm install -g md-to-pdf
```

Then install the backend:

```
$ pip install foliantcontrib.mdtopdf
```

### Usage

```
1 $ foliant make pdf --with mdtopdf
2 Parsing config... Done
3 Applying preprocessor flatten... Done
4 Applying preprocessor mdtopdf... Done
5 Applying preprocessor _unescape... Done
6 Making pdf with md-to-pdf... Done-----
7
8 Result: MyProject.pdf
```

### Config

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

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

```
1 backend_config:
2   mdtopdf:
3     mdtopdf_path: md-to-pdf
4     options:
```

```
5      stylesheet: https://cdnjs.cloudflare.com/ajax/libs/
6      github-markdown-css/2.10.0/github-markdown.min.css
7      body_class: markdown-body
8      css: |-
9          .page-break { page-break-after: always; }
10         .markdown-body { font-size: 11px; }
11         .markdown-body pre > code { white-space: pre-wrap; }
```

**mdtopdf\_path** is the path to `md-to-pdf` executable. Default: `md-to-pdf`  
**options** is a mapping of options which then will be converted into JSON and fed to  
the `md-to-pdf` command. For all possible options consult the [md-to-pdf documentation](#).

## MkDocs

[pypi v1.0.12](#)

[GitHub v1.0.12](#)

# MkDocs Backend for Foliant

The screenshot shows a web page titled "MkDocs". The top navigation bar includes links for "Foliant", "Search", and a GitHub repository icon labeled "foliant-docs/docs 11 Stars · 2 Forks". The left sidebar contains a navigation tree for "Foliant" with sections like "Welcome to Foliant!", "Installation", "Quickstart", "Architecture And Basic Design Concepts", "Tutorials", "Metadata", "Backends", "Agilo", "Confluence", "MdToPdf", "MkDocs", "Pandoc", "Slate", "Preprocessors", "CLI Extensions", "Config Extensions", and "History of Releases". The main content area displays the "MkDocs" page, which describes the backend and provides installation instructions. It includes code snippets for pip installation and command-line usage examples. A "Table of contents" sidebar on the right lists sections such as "Installation", "Usage", "Config", "Preprocessor", and "Troubleshooting".

**Figure 16.** MkDocs site built with Foliant

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

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

## Installation

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

## Usage

Convert Foliant project to MkDocs:

```
1 $ foliant make mkdocs -p my-project✓
2 Parsing config✓
3 Applying preprocessor mkdocs✓
4 Making mkdocs with MkDocs—————
```

```
5
6 Result: My_Project-2017-12-04.mkdocs.src
```

Build a standalone website:

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

Deploy to GitHub Pages:

```
1 $ foliant make ghp -p my-project✓
2 Parsing config✓
3 Applying preprocessor mkdocs✓
4 Making ghp with MkDocs—————
5
6 Result: https://account-name.github.io/my-project/
```

## Config

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

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

```
1 backend_config:
2   mkdocs:
3     mkdocs_path: mkdocs
4     slug: my-awesome_project
5     use_title: true
6     use_chapters: true
7     use_headings: true
8     default_subsection_title: Expand
9     mkdocs.yml:
10       site_name: Custom Title
11       site_url: http://example.com
12       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`. In this case, you don't have to specify `site_name` in `mkdocs.yml` section. If you do, the value from `mkdocs.yml` section has higher priority. If `false`, you must specify `site_name` manually, otherwise MkDocs will not be able to build the site.

Default is `true`.

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

**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 `processors` section manually.

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

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

**mkdocs\_project\_dir\_name** Name of the directory for the generated MkDocs project within the `tmp` directory.

## Troubleshooting

Fenced Code Is Not Rendered in List Items or Blockquotes

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

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

Paragraphs Inside List Items Are Rendered on the Root Level

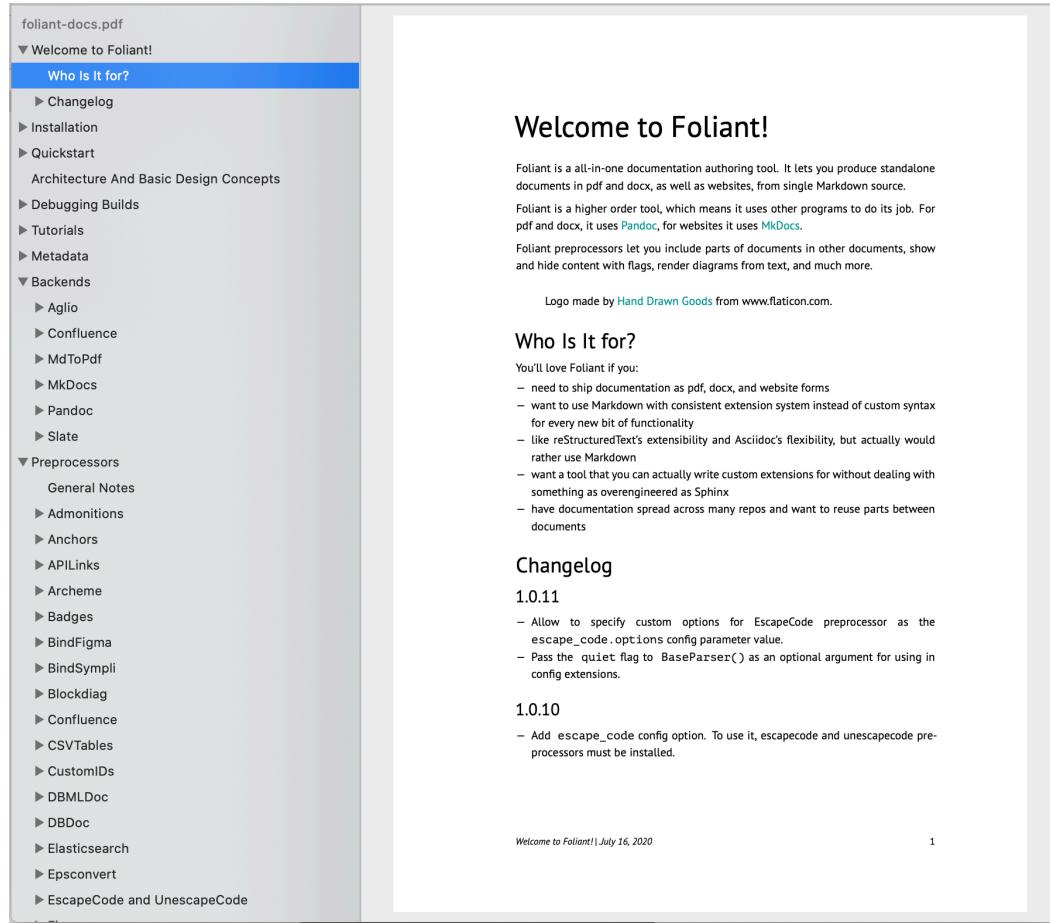
Check if you use **four-space indentation**. [Python Markdown is stern about this point](#).

## Pandoc

[pypi v1.0.11](#)

[GitHub v1.0.11](#)

# Pandoc Backend for Foliant



## 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 custom 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

#### 1.0.11

- Allow to specify custom options for `EscapeCode` preprocessor as the `escape_code.options` config parameter value.
- Pass the `quiet` flag to `BaseParser()` as an optional argument for using in config extensions.

#### 1.0.10

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

Welcome to Foliant! | July 16, 2020

1

**Figure 17.** PDF built with Foliant

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

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

## Installation

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

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

## Usage

Build pdf:

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

Build docx:

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

Build tex (mostly for pdf debugging):

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

## Config

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

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

```
1 backend_config:
2   pandoc:
3     pandoc_path: pandoc
```

```
4     template: !path template.tex
5     vars:
6         ...
7     reference_docx: !path reference.docx
8     params:
9         ...
10    filters:
11        ...
12    markdown_flavor: markdown
13    markdown_extensions:
14        ...
15    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:

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

## 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:

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

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

```
1 backend_config:  
2   pandoc:  
3     ...  
4   params:  
5     pdf_engine: xelatex  
6     listings: false  
7     number_sections: true  
8   ...
```

## Slate

[pypi](#) v1.0.8

[GitHub](#) v1.0.8

# Slate Backend for Foliant

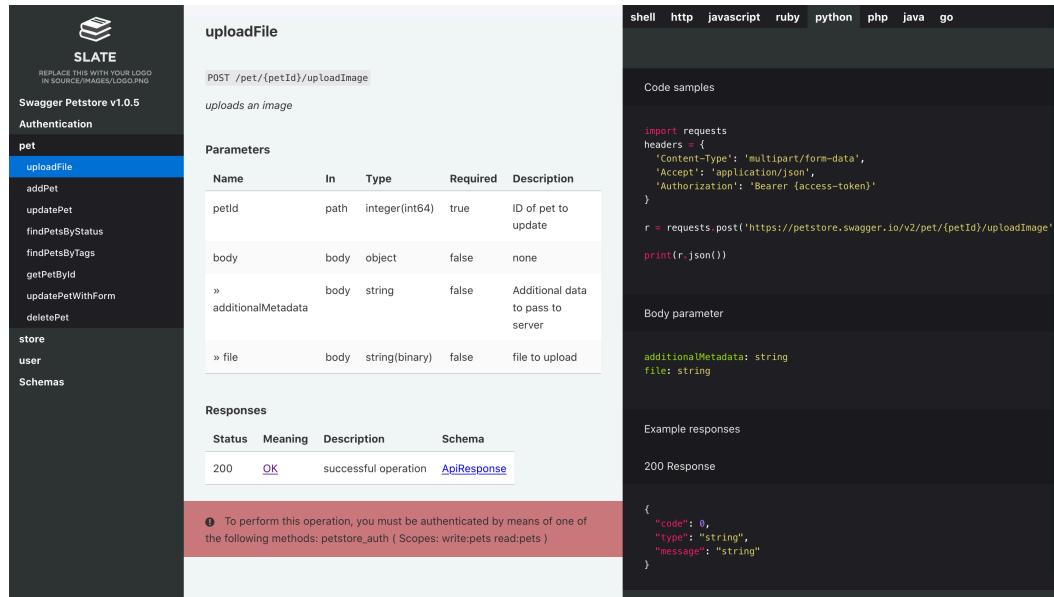


Figure 18. Static site built by Foliant and Slate backend

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
```

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.

```
1 $ bundle exec middleman  
2 == The Middleman is loading
```

```
3 == View your site at ...
4 == Inspect your site configuration at ...
```

## Usage

To convert Foliant project to Slate:

```
1 $ foliant make slate
2 Parsing config... Done
3 Applying preprocessor flatten... Done
4 Applying preprocessor _unescape... Done
5 Making slate... Done————
6
7 Result: My_Project-2018-09-19.src/
```

Build a standalone website:

```
1 $ foliant make site -w slate
2 Parsing config... Done
3 Applying preprocessor flatten... Done
4 Applying preprocessor _unescape... Done————
5
6 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:

```
1 backend_config:
2   slate:
3     shards: data/shards
4     header:
5       title: My API documentation
6       language_tabs:
7         - xml: Response example
8     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:

```
1 slate:  
2     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.

```
1 slate:  
2     shards:  
3         - 'common/shards'  
4         - 'custom/shards'  
5         - '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:

```
1 shards\  
2     source\  
3         stylesheets\  
4             _variables.scss  
5                 screen.css.scss
```

These stylesheets will replace the original ones in the Slate project just before the website is 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:

```
1 preprocessors:  
2     - admonitions
```

### Usage

Add an admonition to your Markdown file:

```
1 !!! warning "optional admonition title"  
2     Admonition text.  
3  
4     May be several paragraphs.
```

Currently supported backends:

- pandoc
- mkdocs\*
- slate

\* for admonitions to work in mkdocs, add `admonition` to the `markdown_extensions` section of your `mkdocs.yml` config:

```
1 backend_config:  
2     mkdocs:  
3         mkdocs.yml:  
4             markdown_extensions:  
5                 - admonition
```

## 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](#) v1.0.4

## 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:

```
1 preprocessors:  
2     - anchors
```

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

```
1 preprocessors:  
2     - anchors:  
3         element: '<span id="{anchor}"></span>'  
4         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:

```
1 ...  
2  
3 <anchor>limitation</anchor>  
4 Some important notice about system limitation.  
5  
6 ...  
7  
8 Don't forget about [limitation](#limitation)!
```

You can also place anchors in the middle of paragraph:

```
1 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.
```

<sup>2</sup>

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

You can place anchors inside tables:

<sup>1</sup> Name	<sup>2</sup> Age	<sup>3</sup> Weight
<sup>4</sup> Max	<sup>5</sup> 17	<sup>6</sup> 60
<sup>7</sup> Jane	<sup>8</sup> 98	<sup>9</sup> 12
<sup>10</sup> John	<sup>11</sup> 10	<sup>12</sup> 40
<sup>13</sup> Katy	<sup>14</sup> 54	<sup>15</sup> 54
<sup>16</sup> Mike	<sup>17</sup> <anchor>Mike</anchor>	<sup>18</sup> 22
<sup>19</sup> Cinty	<sup>20</sup> 25	<sup>21</sup> 42
<sup>22</sup>	<sup>23</sup>	<sup>24</sup>
<sup>25</sup> . . .		
<sup>26</sup>		
<sup>27</sup> Something's	<sup>28</sup> wrong	<sup>29</sup> 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:

```
1 <anchor>!important!</anchor>
2 <anchor>_anchor_</anchor>
3 <anchor>section(1)</anchor>
4 <anchor>section/1/</anchor>
5 <anchor>anchor$1$</anchor>
6 <anchor>about:info</anchor>
7 <anchor>test'1';</anchor>
8 <anchor>якорь></anchor>
9 <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:

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

# APILinks

pypi v1.1.3

## apilinks Preprocessor for Foliant

Preprocessor for replacing API references 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:

```
1 preprocessors:
2   - apilinks:
3     API:
4       My-API:
5         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:

```
1 preprocessors:
2   - apilinks:
3     API:
4       Client-API:
5         url: http://example.com/client/api-docs
6       Admin-API:
7         url: http://example.com/admin/api-docs
```

Now this source:

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

Will be transformed by apilinks into this:

```
1 To authenticate user use API method [GET user/authenticate](http://example.com/client/api-docs/#get-user-authenticate).
2 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:

- <sup>1</sup> Check status of the server through Client API: `Client-API: GET /service/healthcheck`
- <sup>2</sup> 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 referenced. 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 `processors` section in the project config:

```
1 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.

```
1 preprocessors:
2 - apilinks:
3     targets:
4         - site
5     offline: False
6     trim_if_targets:
7         - pdf
8     prefix_to_ignore: Ignore
9     reference:
10        - regex: *ref_pattern
11        only_with_prefixes: false
12        only_defined_prefixes: true
13        output_template: '[{verb} {command}]({url})'
14        trim_template: ``{verb} {command}```
15 API:
16     Client-API:
17         url: http://example.com/api/client
18         default: true
19         header_template: '{verb} {command}'
20     Admin-API:
21         url: http://example.com/api/client
22         header_template: '{command}'
23         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:

```
1 preprocessors:
2   - apilinks:
3     API:
4       Admin-API:
5         url: http://example.com/api/client
6     Client-API:
7       url: http://example.com/api/client
8       default: true
9       header_template: '{verb} {command}'
10    Remote-API:
```

```
11     url: https://remote.net/api-ref/
12     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 should 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:

<sup>1</sup> Unprefixed link with misprint: `GET user/sttus`.

<sup>2</sup> 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:

<sup>1</sup> Unprefixed link with misprint: [GET user/sttus](http://
example.com/api/client/#get-user-sttus).

- 2 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:

- 1 Unprefixed link with misprint: `GET user/sttus`.
- 2 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**

- 1 Prefixed link to the Remote API with a misprint: `Remote-API : GET billling/info`.
- 2 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:

- <sup>1</sup> Prefixed link to the Remote API with a misprint: [GET billling/info](<https://remote.net/api-ref/#get-billling-info>).
- <sup>2</sup> 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:

- <sup>1</sup> Prefixed link to the Remote API with a misprint: `Remote-API : GET billling/info`.
- <sup>2</sup> 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:

```
1 preprocessors:
2   - apilinks:
3     reference:
4       - 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:

```
1 preprocessors:  
2   - apilinks:  
3     reference:  
4       - 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 transformed into links, use your own template. Properties you may use in the template:

property	description	example
url	Full url to the method description	<code>http://example.com/api/#get-user-info</code>
source	Full original reference string	<code>'Client-API: GET user/info'</code>
prefix	Prefix used in the reference	<code>Client-API</code>
verb	HTTP verb used in the reference	<code>GET</code>
command	API command being referenced with endpoint prefix removed	<code>user/info</code>

property	description	example
endpoint_prefix	Endpoint prefix to the API (if <code>endpoint-prefix</code> option is filled in)	<code>/api/v0</code>

## 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:

```

1 ...
2 API:
3   Client-API:
4     header_template: 'Method {command} ({verb})'
5 ...

```

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:

```
1 ...
2 API:
3     Client-API:
4         header_template: '{command}'
5 ...
```

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 <code>endpoint-prefix</code> option is filled in)	/api/v0

## Archeme

[pypi v1.0.2](#)

[GitHub v1.0.2](#)

## Archeme

Archeme preprocessor allows to integrate Foliant with [Archeme](#), a tool for describing and visualizing schemes and diagrams, primarily architectural. Archeme requires [Graphviz](#) to be installed.

Archeme preprocessor finds diagram definitions that are described with Archeme DSL, in source Markdown content, then calls Archeme and Graphviz to draw diagrams, and then replaces the diagram definitions with image references.

## Installation

```
$ pip install foliantcontrib.archeme
```

## Config

To enable the preprocessor, add `archeme` to `processors` section in the project config:

```
1 preprocessors:  
2     - archeme
```

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

```
1 preprocessors:  
2     - archeme:  
3         cache_dir: !path .archemecache  
4         graphviz_paths:  
5             dot: dot  
6             neato: neato  
7             fdp: fdp  
8             action: generate  
9             config_file: null  
10            format: png  
11            targets: []
```

Some values of options specified in the project config may be overridden by tag attributes, see below.

**cache\_dir** Directory to store generated Graphviz sources and drawn diagram images.

**graphviz\_paths** Paths to binaries of Graphviz engines to be used in external commands: `dot`, `neato`, and `fdp`.

**action** Default action. Used when the respective `action` tag attribute is not specified explicitly, see below. Available values are: `generate` (default), and `merge` ([see descriptions in Archeme documentation](#)).

**config\_file** Path to default config file. May be overridden with the value of the respective `config_file` tag attribute, see below. Config file usually defines common settings of multiple diagrams, it's recommended but not strictly required. By default, no config file is used.

**format** Format of the output image. May take any value supported by Graphviz, but note that drawn images are used within Markdown content that will be rendered by one or another backend. Preferred values are: `png` (default), and `svg`.

**targets** Allowed targets for the preprocessor. If not specified (by default), the pre-processor applies to all targets. Limitation of available targets may be useful when it's needed to build a certain Foliant project in different ways with various settings, e.g. as a stand-alone documentation (for example, with the `site` target), and as a part of a documentation that combines several Foliant projects (in this case the `pre` target is usually used).

## Usage

To insert an Archeme diagram definition into your Markdown source, enclose it between `<archeme>...</archeme>` tags:

```
1 <archeme>
2 structure:
3   - node:
4     id: first
5   - node:
6     id: second
7 edges:
8   - tail: first
9     head: second
10 </archeme>
```

You may use optional tag attributes:

- `id`—to specify an unique ID of the diagram that may be used for merging multiple diagram definitions;
- `action`—action that should be applied to the diagram definition; the available values are `generate` and `merge`; this attribute overrides the respective `action config` option;
- `config_file`—path to a specific config file for the certain diagram definition; this attribute overrides the respective `config_file config` option;
- `format`—output image format for the certain diagram definition; this attribute overrides the respective `format config` option.

## Examples

Diagram definition with explicitly specified ID, config file, and output format:

```
1 <archeme id="one" config_file="!project_path another_config.yml" format="svg">
2   structure:
3     - node:
4       id: first
5     - node:
6       id: second
7   edges:
8     - tail: first
9       head: second
10 </archeme>
```

Archeme DSL definition that prescribes to combine two modules with explicitly specified IDs:

```
1 <archeme action="merge">
2   structure:
3     - module:
4       id: one
5     - module:
6       id: two
7 </archeme>
```

Note that the `file` and `description` module parameters in Archeme DSL work as usual. If you need to combine the diagrams that are identified within the current Foliant project by using `<archeme id="...">` tags, you should omit the `file` and `description` module parameters in your combined diagrams definitions.

# 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 `processors` section in the project config:

```
1 preprocessors:  
2     - badges
```

The preprocessor has a number of options:

```
1 preprocessors:  
2     - badges:  
3         server: 'https://img.shields.io'  
4         as_object: true  
5         add_link: true  
6         vars:  
7             jira_path: localhost:3000/jira  
8             package: foliant  
9             # badge look parameters  
10            style: flat-square  
11            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:

```
1 preprocessors:  
2   - badges:  
3     vars:  
4       jira: https/issues.apache.org/jira
```

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

```
1 <badge>jira/issue/${jira}/kafka-2896.svg</badge>
2
3 Description of the issue goes here. But it's not the only
one.
4
5 <badge>jira/issue/${jira}/KAFKA-7951.svg</badge>
6
7 Description of the second issue.
```

## BindFigma

[pypi](#) v1.0.3

[GitHub](#) v1.0.3

## BindFigma

BindFigma is a preprocessor that downloads and optionally resizes design layout images from [Figma](#), and binds these images with the documentation project.

The preprocessor uses [Figma REST API](#) to get URLs of images to download. To use the preprocessor, you should get an [access token](#) for it via your Figma account.

If you need to resize downloaded images, you should install [ImageMagick](#).

### Installation

```
$ pip install foliantcontrib.bindfigma
```

### Config

To enable the preprocessor, add `bindfigma` to `processors` section in the project config:

```
1 preprocessors:
2     - bindfigma
```

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

```
1 preprocessors:
2   - bindfigma:
3     cache_dir: !path .bindfigmacache
4     api_caching: disabled
5     convert_path: convert
6     caption: ''
7     hyperlinks: true
8     multi_delimeter: '\n\n'
9     resize: null
10    access_token: null
11    file_key: null
12    ids: null
13    scale: null
14    format: null
15    svg_include_id: null
16    svg_simplify_stroke: null
17    use_absolute_bounds: null
18    version: null
```

Some values of options specified in the project config may be overridden by tag attributes, see below.

**cache\_dir** Directory to store cached API responses, downloaded and resized images.

**api\_caching** API responses caching mode. Available values: `disabled`—switch off unconditionally; `enabled`—switch on unconditionally; `env`—switch on only if the `FOLIANT_FIGMA_CACHING` environment variable is set, otherwise switch off. If this mode is switched on, the preprocessor caches Figma API responses locally and uses cached data instead of performing API request. In this case, Figma node updating without changing API URL may not take effect.

**convert\_path** Path to `convert` binary, a part of ImageMagick. If resizing is not needed, ImageMagick will not be used.

**caption** Caption of images. The `{{image_id}}` placeholder in the caption will be replaced with Figma node ID.

**hyperlinks** Flag that tells the preprocessor to wrap image references into hyperlinks to related Figma URLs.

**multi\_delimeter** String that should separate multiple image references.

**resize** Width of resulting images in pixels. If not specified, resizing is not performed.

**access\_token** Access token that you can generate in your Figma account.

**file\_key** ID of the Figma file.

**ids** One or more IDs of nodes in the Figma file. May be specified as a list or as a comma-separated string.

**scale, format, svg\_include\_id, svg\_simplify\_stroke, use\_absolute\_bounds, version**  
Query parameters to use in Figma API requests, see descriptions in [Figma API documentation](#).

## Usage

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

```
1 '
2 Heres an image from Figma:
3
4 <figma caption="An optional caption" resize="300" file_key="
ABC" ids="node1,node2,node3"></figma>
```

You may use tag attributes to override the values of the project config options with the same names. All the options excepting `cache_dir`, `api_caching` and `convert_path` may be overridden in this way.

BindFigma preprocessor will replace such statements with local image references. If `ids` refers to more than one image, a set of image references will be generated. Multiple image references will be separated with the string specified as `multi_delimeter`.

## BindSympli

[pypi v1.0.14](#)

[GitHub v1.0.14](#)

## 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 `processors` section in the project config:

```
1 preprocessors:  
2     - bindsympli
```

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

```
1 preprocessors:  
2     - bindsympli:  
3         get_sympli_img_urls_path: get_sympli_img_urls.js  
4         wget_path: wget  
5         convert_path: convert  
6         cache_dir: !path .bindsymplocache  
7         sympli_login: ''  
8         sympli_password: ''  
9         image_width: 800  
10        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:

```
1 '
2 Heres an image from Sympli:
3
4 <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

[pypi v1.0.5](#)

[GitHub v1.0.5](#)

# Blockdiag Preprocessor for Foliant

[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 `processors` section in the project config:

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

The preprocessor has a number of options:

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

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

### Note

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

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

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

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

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

## Usage

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

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

```
19     browser <-- webserver;
20 }
21 </seqdiag>
```

To set a caption, use `caption` option:

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

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

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

## Confluence

[pypi](#) v0.6.12

[GitHub](#) v0.6.12

Confluence preprocessor allows inserting content from Confluence server into your Foliant project.

## Installation

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

## Config

To enable the preprocessor, add `confluence` to `processors` section in the project config:

```
1 preprocessors:  
2     - confluence
```

The preprocessor has a number of options:

```
1 preprocessors:  
2     - confluence:  
3         passfile: confluence_secrets.yml  
4         host: https://my_confluence_server.org  
5         login: user  
6         password: user_password  
7         space_key: "~user"  
8         pandoc_path: pandoc
```

**passfile** Path to YAML-file holding credentials. See details in [Supplying Credentials](#) section. Default: `confluence_secrets.yml`

**host** **Required** Host of your confluence server. If not stated – it would be taken from Confluence backend config.

**login** Login of the user who has permissions to create and update pages. If login is not supplied, it would be taken from backend config, or prompted during the build.

**password** Password of the user. If password is not supplied, it would be taken from backend config, or prompted during the build.

**space\_key** The space key where the page titles will be searched for.

**pandoc\_path** Path to Pandoc executable (Pandoc is used to convert Confluence content into Markdown).

## Usage

Add a `<confluence></confluence>` tag at the position in the document where the content from Confluence should be inserted. The page is defined by its `id` or

`title`. If you are specifying page by title, you will also need to set `space_key` either in tag or in the preprocessor options.

```
1 The following content is imported from Confluence:  
2  
3 <confluence id="12345"></confluence>  
4  
5 This is from Confluence too, but determined by page title (  
 space key is defined in preprocessor config):  
6  
7 <confluence title="My Page"></confluence>  
8  
9 Here we are overriding space_key:  
10  
11 <confluence space_key="ANOTHER_SPACE" title="My Page"></  
 confluence>
```

## Supplying Credentials

There are two ways to supply credentials for your confluence server.

### 1. In foliant.yml

The most basic way is just to put credentials in foliant.yml:

```
1 backend_config:  
2     confluence:  
3         host: https://my_confluence_server.org  
4         login: user  
5         password: pass
```

It's not very secure because foliant.yml is usually visible to everybody in your project's git repository.

### 2. Using passfile

Alternatively, you can use a passfile. Passfile is a yaml-file which holds all your passwords. You can keep it out from git-repository by storing it only on your local machine and production server.

To use passfile, add a `passfile` option to foliant.yml:

```
1 backend_config:  
2     confluence:  
3         host: https://my_confluence_server.org  
4         passfile: confluence_secrets.yaml
```

The syntax of the passfile is the following:

```
1 hostname:  
2     login: password
```

For example:

```
1 https://my_confluence_server.org:  
2     user1: wFwG34uK  
3     user2: MEUeU3b4  
4 https://another_confluence_server.org:  
5     admin: adminpass
```

If there are several records for a specified host in passfile (like in the example above), Foliant will pick the first one. If you want specific one of them, add the login parameter to your foliant.yml:

```
1 backend_config:  
2     confluence:  
3         host: https://my_confluence_server.org  
4         passfile: confluence_secrets.yaml  
5         login: user2
```

## 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 `processors` section in the project config:

```
1 preprocessors:  
2   - csvtables
```

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

```
1 preprocessors:  
2   - csvtables:  
3     delimiter: ';'  
4     padding_symbol: ' '  
5     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.

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

Or in external `file.csv`.

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

You can reassign setting for certain csv tables.

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

## Example

Usage section will be converted to this:

You can place csv data in `csvtable` tag.

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

Or in external `file.csv`.

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

You can reassign setting for certain csv tables.

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

3 | *Datum 1* | *Datum 2* | *Datum 3* | *Datum 4* | *Datum
5* |
4 | *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 `processors` section in the project config:

```
1 preprocessors:  
2     - customids
```

The preprocessor supports the following options:

```
1     - customids:  
2         stylesheet_path: !path customids.css  
3         targets:  
4             - pre  
5             - mkdocs  
6             - site  
7             - 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 pre-processor applies to all targets.

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

```
1 # First Heading {#custom_id_for_first_heading}  
2  
3 A paragraph.  
4  
5 ## Second Heading {#custom_id_for_second_heading}  
6  
7 Some another paragraph.
```

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

```
1 <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>
```

```

2
3 <h1>First Heading</h1>
4
5 <p>A paragraph.</p>
6
7 <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>
8
9 <h2>Second Heading</h2>
10
11 <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:

```

1 [Link to Heading 1](#custom_id_for_first_heading)
2
3 [Link to Heading 2 in some document at the current site](/some/page/#custom_id_for_second_heading)
4
5 [Link to some heading with custom ID at an external site](:https://some.site/path/to/the/page/#some_custom_id)

```

## DBMLDoc

[pypi](#) v0.2.4

[GitHub](#) v0.2.4

# DBML Docs Generator for Foliant

This preprocessor generates Markdown documentation from [DBML](#) specification files. It uses [PyDBML](#) for parsing DBML syntax and [Jinja2](#) templating engine for generating Markdown.

## Installation

```
$ pip install foliantcontrib.dbmldoc
```

## Config

To enable the preprocessor, add `dbmldoc` to `processors` section in the project config:

```
1 preprocessors:  
2     - dbmldoc
```

The preprocessor has a number of options:

```
1 preprocessors:  
2     - dbmldoc:  
3         spec_url: http://localhost/scheme.dbml  
4         spec_path: scheme.dbml  
5         doc: true  
6         scheme: true  
7         template: dbml.j2  
8         scheme_template: scheme.j2
```

**spec\_url** URL to DBML spec file. If it is a list – preprocessor uses the first url which works.

**spec\_path** Local path to DBML spec file (relative to project dir).

If both url and path params are specified – preprocessor first tries to fetch spec from url, and only if that fails looks for the file on the local path.

**doc** If `true` – documentation will be generated. Set to `false` if you only want to draw a schema of the database. Default `true`

**scheme** If `true` – the platuml code for database schema will be generated. Default `true`

**template** 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: dbml.j2

**scheme\_template** Path to jinja-template for generating planuml code for the database schema. 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: scheme.j2

## Usage

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

```
1 # Introduction
2
3 This document contains the automatically generated
4 documentation of our Database schema.
5 <dbmldoc></dbmldoc>
```

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

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

```
1 # Introduction
2
3 Introduction text for API documentation.
4
5 <dbmldoc spec_url="http://localhost/schema.dbml"
6           template="dbml.j2"
7           scheme="false">
8 </dbmldoc>
9
10 # Database schema
11
12 And here goes a visual diagram of our database:
13
```

```
14 <dbmldoc doc="false" scheme="true">
15 </dbmldoc>
```

Note that template path in tag is stated **relative to the markdown file**.

Tag parameters have the highest priority.

This way you can put your database description in one place and its diagram in the other (like in the example above). Or you can even have documentation from several different DBML spec files in one Foliant project.

## Customizing output

The output markdown is generated by the [Jinja2](#) template. Inside the template all data from the parsed DBML file is available under the `data` variable. It is in fact a `PyDBMLParseResults` object, as returned by [PyDBML](#) (see the docs to find out which attributes are available).

To customize the output create a template which suits your needs. Then supply the path to it in the `template` parameter. Same goes for the scheme template, which is defined in the `scheme_template` parameter.

If you wish to use the default template as a starting point, build the foliant project with `dbmldoc` preprocessor turned on. After the first build the default templates will appear in your foliant project dir under the names `dbml.j2` and `scheme.j2`.

# DBDoc

[pypi](#) v0.1.2

[GitHub](#) v0.1.2

# Database Documentation Generator for Foliant

The screenshot shows a database documentation page for the 'REGIONS' table. On the left, there's a sidebar with a logo and a 'Tables' section containing links to 'COUNTRIES', 'DEPARTMENTS', 'EMPLOYEES', 'JOBS', 'JOB\_HISTORY', 'LOCATIONS', 'REGIONS' (which is highlighted in blue), 'Views', 'Functions', 'Triggers', and 'Database Scheme'. The main content area has a header 'REGIONS' with a sub-header 'Regions table contains region numbers and names. Contains 4 rows; references with the Countries table.' Below this is a table with columns: column, nullable, type, descr, and fkey. It lists two rows: 'REGION\_ID' (nullable N, type NUMBER, descr: Primary key of regions table) and 'REGION\_NAME' (nullable Y, type VARCHAR2, descr: Names of regions. Locations are in the countries of these regions.). Under 'Views', there's a section for 'EMP\_DETAILS\_VIEW' with 'Name: EMP\_DETAILS\_VIEW' and 'Schema: HR'. To the right, there's a large black rectangular area.

Static site on the picture was built with [Slate](#) backend together with [DBDoc preprocessor](#)

This preprocessor generates simple documentation based on the structure of the database. It uses [Jinja2](#) templating engine for customizing the layout and [PlantUML](#) for drawing the database scheme.

Currently supported databases:

- [PostgreSQL](#),
- [Oracle](#).

## Installation

### Prerequisites

DBDoc generates documentation by querying database structure. That's why you will need client libraries installed on your computer before running the preprocessor.

PostgreSQL will be installed automatically with the preprocessor.

But Oracle libraries are proprietary, so we cannot include them even in our [Docker distribution](#). So, if you are planning on using DBDoc to document Oracle databases, first install the [Instant Client](#).

If you search the web, you can find ways to install Oracle Instant Client inside your Docker image, just saying.

## Preprocessor

```
$ pip install foliantcontrib.dbdoc
```

## Config

To enable the preprocessor, add `dbdoc` to `processors` section in the project config:

```
1 preprocessors:
2     - dbdoc
```

The preprocessor has a number of options:

```
1 preprocessors:
2     - dbdoc:
3         host: localhost
4         port: 5432
5         dbname: postgres
6         user: postgres
7         password: ''
8         doc: True
9         scheme: True
10        filters:
11            ...
12        doc_template: dbdoc.j2
13        scheme_template: scheme.j2
14        components:
15            - tables
16            - functions
17            - triggers
```

**host** Database host address. Default: `localhost`

**port** Database port. Default: 5432 for `pgsql`, 1521 for `Oracle`.

**dbname** PostgreSQL database name. Default: `postgres` for `pgsql`, `orcl` for `oracle`.

**user** PostgreSQL user name. Default: `postgres` for `pgsql`, `hr` for `oracle`.

**password** PostgreSQL user password. Default: `postgres` for `pgsq`, `oracle` for `oracle`.

**doc** If `true` – documentation will be generated. Set to `false` if you only want to draw a scheme of the database. Default: `true`

**scheme** If `true` – the platuml code for database scheme will be generated. Default: `true`

**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. If not supplied – default template would be used.

**scheme\_template** Path to jinja-template for scheme. Path is relative to the project directory. If not supplied – default template would be used.

**components** List of components to be added to documentation. If not supplied – everything will be added. Use to exclude some parts of documentation. Available components: `'tables'`, `'views'`, `'functions'`, `'triggers'`.

## Usage

DBDoc currently supports two database engines: Oracle and PostgreSQL. To generate Oracle database documentation, add an `<oracle></oracle>` tag to a desired place of your chapter.

```
1 # Introduction
2
3 This document contains the most awesome automatically
   generated documentation of our marvellous Oracle database.
4
5 <oracle></oracle>
```

To generate PostgreSQL database documentation, add a `<pgsql></pgsql>` tag to a desired place of your chapter.

```
1 # Introduction
2
3 This document contains the most awesome automatically
   generated documentation of our marvellous Oracle database.
4
5 <pgsql></pgsql>
```

Each time the preprocessor encounters one of the mentioned tags, 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:

```
1 # Introduction
2
3 Introduction text for database documentation.
4
5 <oracle scheme="true"
6     doc="false"
7     host="11.51.126.8"
8     port="1521"
9     dbname="mydb"
10    user="scott"
11    password="tiger">
12 </oracle>
```

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). It also allows you to put documentation and scheme for your database separately by switching on/off **doc** and **scheme** params in tags.

## Filters

You can add filters to exclude some tables from the documentation. dbdocs 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:

```
1 preprocessors:
2     - dbdoc:
3         filters:
4             eq:
5                 schema: public
6             regex:
7                 table_name: 'main_.+'
```

or in tag options using the same yaml-syntax:

```
1 <pgsql filters=""
2   eq:
3     schema: public
4   regex:
5     table_name: 'main_.+'>
6 </pgsql>
```

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	~, REGEX_LIKE	matches regular expression	literal
not_regex	!~, NOT REGEX_LIKE	does not match regular expression	literal

List of currently supported filtering fields:

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

The syntax for using filters in configuration files is following:

```
1 filters:
2   <operator>:
3     <field>: value
```

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

```
1 filters:
2   in:
3     schema:
4       - public
5       - 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 dbdoc will use default templates.

If you wish to create your own template, the default ones may be a good starting point.

- [Default Oracle doc template](#).
- [Default Oracle scheme template](#).
- [Default PostgreSQL doc template](#).
- [Default PostgreSQL scheme template](#).

## Elasticsearch

This extension allows to integrate Foliant-managed documentation projects with [Elasticsearch](#) search engine.

The main part of this extension is a preprocessor that prepares data for a search index. Also this extension provides a simple working example of a client-side Web application that may be used to perform searching. By editing HTML, CSS and JS code you may customize it according to your needs.

### Installation

To install the preprocessor, run the command:

```
$ pip install foliantcontrib.elasticsearch
```

To use an example of a client-side Web application for searching, download [these HTML, CSS, and JS files](#) and open the file `index.html` in your Web browser.

### Config

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

```
1 preprocessors:
```

```
2     - elasticsearch
```

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

```
1 preprocessors:
2     - elasticsearch:
3         es_url: 'http://127.0.0.1:9200/'
4         index_name: ''
5         index_copy_name: ''
6         index_properties: {}
7         actions:
8             - delete
9             - create
10        use_chapters: true
11        format: plaintext
12        escape_html: true
13        url_transform:
14            - '/?index\.\md$': '/'
15            - '\.\md$': '/'
16            - '^([^\/]*)': '/\g<1>'
17        require_env: false
18        targets: []
```

**es\_url** Elasticsearch API URL.

**index\_name** Name of the index. Your index must have an explicitly specified name, otherwise (by default) API URL will be invalid.

**index\_copy\_name** Name of the index copy when the `copy` action is used; see below. If the `index_copy_name` is not set explicitly, and if the `index_name` is specified, the `index_copy_name` value will be formed as the `index_name` value with the `_copy` string appended to the end.

**index\_properties** Settings and other properties that should be used when creating an index. If not specified (by default), the default Elasticsearch settings will be used. More details are described below.

**actions** Sequence of actions that the preprocessor should perform. Available item values are: `delete`, `create`, `copy`. By default, the actions `delete` and `create` are performed since in most cases it's needed to remove and then

fully rebuild the index. The `copy` action is used to duplicate an index, i.e to create a copy of the index `index_name` with the name `index_copy_name`. This action may be useful when a common search index is created for multiple Foliant projects, and the index may remain incomplete during for a long time during their building. The `copy` action is not atomic. To perform it, the preprocessor:

- marks the source index `index_name` as read-only;
- deletes the target index `index_copy_name` if it exists;
- clones the source index `index_name` and thereby creates the target index `index_copy_name`;
- unmarks the source index `index_name` as read-only;
- also unmarks the target index `index_copy_name` as read-only, since the target index inherits the settings of the source one.

**use\_chapters** If set to `true` (by default), the preprocessor applies only to the files that are mentioned in the `chapters` section of the project config. Otherwise, the preprocessor applies to all of the files of the project.

**format** Format that the source Markdown content should be converted to before adding to the index; available values are: `plaintext` (by default), `html`, `markdown` (for no conversion).

**escape\_html** If set to `true` (by default), HTML syntax constructions in the content converted to `plaintext` will be escaped by replacing `&` with `&amp;`, `<` with `<lt;`, `>` with `>t;`, and `"` with `"`.

**url\_transform** Sequence of rules to transform local paths of source Markdown files into URLs of target pages. Each rule should be a dictionary. Its data is passed to the `re.sub()` method: key as the `pattern` argument, and value as the `repl` argument. The local path (possibly previously transformed) to the source Markdown file relative to the temporary working directory is passed as the `string` argument. The default value of the `url_transform` option is designed to be used to build static websites with MkDocs backend.

**require\_env** If set to `true`, the `FOLIANT_ESPRESSO` environment variable must be set to allow the preprocessor to perform any operations with Elasticsearch index. This flag may be useful in CI/CD jobs.

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

## Usage

The preprocessor reads each source Markdown file and generates three fields for indexing:

- `url`—target page URL;
- `title`—document title, it's taken from the first heading of source Markdown content;
- `content`—source Markdown content, optionally converted into plain text or HTML.

When all the files are processed, the preprocessor calls Elasticsearch API to create the index.

Optionally the preprocessor may call Elasticsearch API to delete previously created index.

By using the `index_properties` option, you may override the default Elasticsearch settings when creating an index. Below is an example of JSON-formatted value of the `index_properties` option to create an index with Russian morphology analysis:

```
1 {
2     "settings": {
3         "analysis": {
4             "filter": {
5                 "ru_stop": {
6                     "type": "stop",
7                     "stopwords": "_russian_"
8                 },
9                 "ru_stemmer": {
10                     "type": "stemmer",
11                     "language": "russian"
12                 }
13             },
14             "analyzer": {
15                 "default": {
16                     "tokenizer": "standard",
17                     "filter": [
18                         "lowercase",
19                         "ru_stop",
```

```
20                     "ru_stemmer"
21                 ]
22             }
23         }
24     }
25 }
```

You may perform custom search requests to Elasticsearch API.

The [simple client-side Web application example](#) that is provided as a part of this extension, performs requests like this:

```
1 {
2     "query": {
3         "multi_match": {
4             "query": "foliant",
5             "type": "phrase_prefix",
6             "fields": [ "title^3", "content" ]
7         }
8     },
9     "highlight": {
10        "fields": {
11            "content": {}
12        }
13    },
14    "size": 50
15 }
```

Search results may look like that:

The screenshot shows a web browser window with the URL <http://127.0.0.1/>. The search bar contains the text "foliant". Below the search bar, the text "Results: 42" is displayed. The main content area features a heading "Welcome to Foliant!" and a sub-heading "Page URL: <http://localhost/>". A code block follows:

```
Welcome to Foliant!
Foliant is a all-in-one documentation authoring tool.

Foliant is a higher order tool, which means it uses other programs to do its job.

Foliant preprocessors let you include parts of documents in other documents, show and hide content with

You'll love Foliant if you:

need to ship documentation as pdf, docx, and website forms
want to use Markdown

Add pre backend with pre target that applies the preprocessors from the config and returns a Foliant
```

## Documenting API with Foliant

Page URL: <http://localhost/tutorials/api/>

```
Documenting API with Foliant
In this tutorial we will learn how to use Foliant to generate documentation

Creating project
Let's create Foliant project. The easiest way is to use foliant init command.

Creating project
Let's create Foliant project. The easiest way is to use foliant init command.

```shell
foliant make site --with slate
  Parsing config...
The easiest way is to use foliant init command.
```

**Figure 19.** Search Results

If you use self-hosted instance of Elasticsearch, you may need to configure it to append [CORS](#) headers to HTTP API responses.

## 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 `processors` section in the project config:

```
1 preprocessors:  
2     - epsconvert
```

The preprocessor has a number of options:

```
1 preprocessors:  
2     - epsconvert:  
3         convert_path: convert  
4         cache_dir: !path .epsconvertcache  
5         image_width: 0  
6         targets:  
7             - pre  
8             - mkdocs  
9             - site  
10            - 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

[pypi](#) v1.0.3

[GitHub](#) v1.0.2

## 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.escapecode
```

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.

```
1 >    **Note**  
2 >
```

```
3 > 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:

```
1 title: My Awesome Project  
2  
3 chapters:  
4     - index.md  
5     ...  
6  
7 escape_code: true  
8  
9 preprocessors:  
10    ...  
11    - includes  
12    ...  
13 ...
```

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.

In more complicated case, you may pass some custom options to `EscapeCode` preprocessor:

```
1 escape_code:  
2     options:  
3     ...
```

Custom options available in `EscapeCode` since version 1.0.2. Foliant core supports passing custom options to `EscapeCode` preprocessor as the value of `escape_code.options` parameter since version 1.0.11. Options are described below.

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 want to use the `escape_code` option and call the preprocessors explicitly:

```
1 preprocessors:  
2     - escapecode          # usually the first list item  
3     ...  
4     - unescapecode        # usually the last list item
```

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

```
1 preprocessors:  
2     - escapecode:  
3         cache_dir: !path .escapecodecache  
4     ...  
5     - unescapecode:  
6         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.

## Config

Since version 1.0.2, EscapeCode preprocessor supports the option `actions` in addition to `cache_dir`.

The value of `actions` options should be a list of acceptable actions. By default, the following list is used:

```
1 actions:  
2     - normalize  
3     - escape:
```

```
4      - fence_blocks
5      - pre_blocks
6      - inline_code
```

This default list may be overridden. For example:

```
1 actions:
2   - normalize
3   - escape:
4     - fence_blocks
5     - inline_code
6     - tags:
7       - plantuml
8       - seqdiag
9     - comments
```

Meanings of parameters:

- `normalize`—perform normalization;
- `escape`—perform escaping of certain types of raw content:
  - `fence_blocks`—fence code blocks;
  - `pre_blocks`—pre code blocks;
  - `inline_code`—inline code;
  - `comments`—HTML-style comments, also usual for Markdown;
  - `tags`—content of certain tags with the tags themselves, for example `plantuml` for `<>plantuml>...</plantuml>`.

## Usage

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

```
1 # Heading
2
3 Text that contains some `inline code`.
4
5 Below is a fence code block, language is optional:
6
7 ````python
8 import this
9 ````
```

```

10
11 One more fence code block:
12
13 ~~~
14 # This is a comment that should not be interpreted as a
   heading
15
16 print('Hello World')
17 ~~~
18
19 And this is a pre code block:
20
21     mov dx, hello;
22     mov ah, 9;
23     int 21h;

```

The preprocessor EscapeCode with default behavior will do the following replacements:

```

1 # Heading
2
3 Text that contains some <>escaped hash="2
  bb20aeb00314e915ecfefd86d26f46a"></escaped>.
4
5 Below is a fence code block, language is optional:
6
7 <>escaped hash="15e1e46a75ef29eb760f392bb2df4ebb"></escaped>
8
9 One more fence code block:
10
11 <>escaped hash="91c3d3da865e24c33c4b366760c99579"></escaped>
12
13 And this is a pre code block:
14
15 <>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`:

```
1 ````python
2 import this
3 ````
```

## Flags

pypi v1.0.2

GitHub v1.0.2

## Conditional Blocks for Foliant

This preprocessor 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 processor by adding it to `processors`:

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

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

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

```
5      - bar
```

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

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

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

### Hint

To emulate a particular target or backend with a flag, use the special flags `target:FLAG` and `backend:FLAG` where `FLAG` is your target or backend:

```
$ FOLIANT_FLAGS="target:pdf, backend:pandoc, spam  
"
```

## Usage

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

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

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

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

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

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

```
3 </if>
```

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

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

## Flatten

[pypi](#) v1.0.7

[GitHub](#) v1.0.7

### Project Flattener for Foliant

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

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

#### Installation

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

#### Config

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

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

```
1 preprocessors:
```

```
2     - flatten
```

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

```
1 preprocessors:
2     - flatten:
3         flat_src_file_name: __all__.md
4         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

[pypi v1.0.0](#)

[GitHub v1.0.0](#)

## Glossary collector for Foliant

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 `processors` section in the project config.

```
1 preprocessors:  
2   - glossary
```

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

```
1 preprocessors:  
2   - glossary:  
3     term_definitions: 'term_definitions.md'  
4     definition_mark: ':'  
5     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**

```
1 ...  
2 chapters:  
3   - text.md  
4  
5 preprocessors:  
6 ...
```

```
7     - includes  
8     - glossary  
9 ...
```

### **term\_definitions.md**

```
1 # Glossary  
2  
3 <include nohead="true">  
4     $https://git.repo/repo_name_1$src/glossary_1.md  
5 </include>  
6  
7 <include nohead="true">  
8     $https://git.repo/repo_name_2$src/glossary_2.md  
9 </include>
```

#### **glossary\_1.md from repo\_1**

```
1 # Glossary  
2  
3 Term 1  
4  
5 :    Definition 1  
6  
7 Term 2  
8  
9 :    Definition 2  
10  
11 Term 3  
12  
13 :    Definition 3
```

#### **glossary\_2.md from repo\_2**

```
1 # Glossary  
2  
3 Term 4  
4  
5 :    Definition 4
```

```
6
7      { some code, part of Definition 4 }
8
9      Third paragraph of definition 4.
10
11 Term 5
12
13 : Definition 5
```

### **text.md**

```
1 # Main chapter
2
3 Some text.
4
5 # Glossary
6
7 : Term 1
8
9 : Term 4
10
11 : Term 2
```

### **all\_.md**

```
1 # Main chapter
2
3 Some text.
4
5 # Glossary
6
7 Term 1
8
9 : Definition 1
10
11
12 Term 4
13
14 : Definition 4
```

```
15
16     { some code, part of Definition 4 }
17
18     Third paragraph of definition 4.
19
20
21 Term 2
22
23 :    Definition 2
```

## Graphviz

pypi v1.1.3

github v1.1.3

## 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 `processors` section in the project config:

```
1 preprocessors:
2     - graphviz
```

The preprocessor has a number of options:

```

1 preprocessors:
2     - graphviz:
3         cache_dir: !path .diagramscache
4         graphviz_path: dot
5         engine: dot
6         format: png
7         as_image: true
8         params:
9             ...

```

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

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:

```

1 preprocessors:
2     - graphviz:
3         params:
4             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:

```
1 '
2 Heres a diagram:
3
4 <graphviz>
5     a -> b
6 </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.

```
1 Diagram with a caption:
2
3 <graphviz caption="Deployment diagram"
4         params="Earrowsize: 0.5"
5         src="diags/sample.gv">
6 </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

pypi v1.0.8

## 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 `processors` section in the project config:

```
1 preprocessors:  
2     - history
```

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

```
1 - history:  
2     repos: []  
3     revision: master  
4     name_from_readme: false  
5     readme: README.md  
6     from: changelog  
7     merge_commits: true  
8     changelog: changelog.md  
9     source_heading_level: 1  
10    target_heading_level: 1  
11    target_heading_template: '[%date%] [%repo%](%link%) %  
12      version%'  
13    date_format: year_first  
14    limit: 0  
15    rss: false
```

```
15     rss_file: rss.xml
16     rss_title: 'History of Releases'
17     rss_link: ''
18     rss_description: ''
19     rss_language: en-US
20     rss_item_title_template: '%repo% %version%'
```

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

Example:

```
1 repos:
2   - https://github.com/foliant-docs/foliant.git
3   - 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:

<sup>1</sup> Some optional content here.

```
2
3 <history></history>
4
5 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:

```
1 <history
2     repos="https://github.com/foliant-docs/foliantcontrib.
3     mkdocs.git"
4     revision="develop"
5     limit="5"
6     rss="true"
7     rss_file="some_another.xml"
8     ...
9 >
9 </history>
```

## ImageMagick

[pypi](#) v1.0.2

[GitHub](#) v1.0.2

## ImageMagick Preprocessor

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 `processors` section in the project config:

```
1 preprocessors:  
2     - imagemagick
```

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

```
1 preprocessors:  
2     - imagemagick:  
3         convert_path: convert  
4         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.

```
1 <magick command_params="-resize 600 -background Orange label  
:'Picture' +swap -gravity Center -append" output_format="jpg  
">  
2 (leading exclamation mark here)[Optional Caption](image.eps)  
3 </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:

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

```
1 preprocessors:
2   - imgcaptions
```

The preprocessor supports the following options:

```
1   - imgcaptions:
2     stylesheet_path: !path imgcaptions.css
3     template: <p class="image_caption">{caption}</p>
4     targets:
5       - pre
6       - mkdocs
7       - site
8       - 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.

**template** Template string representing the HTML tag of the caption to be placed after the image. The template should contain the `{caption}` variable that will be replaced with the image caption. Default: `<p class="image_caption">{caption}</p>`.

**targets** Allowed targets for the preprocessor. If not specified (by default), the pre-processor 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:

```
1 <p></p>  
2 <p class="image_caption">My Picture</p>
```

(Note that ImgCaptions preprocessor does not convert Markdown syntax into HTML; it only inserts HTML tags like `<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 `processors` section in the project config:

```
1 preprocessors:  
2   - imgconvert
```

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

```
1 preprocessors:  
2   - imgconvert:  
3     convert_path: convert
```

```
4     cache_dir: !path .imgconvertcache
5     image_width: 0
6     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:

```
1 formats:
2     eps:
3         targets:
4             - site
5     svg:
6         targets:
7             - docx
```

Formats should be named in lowercase.

## Includes

[pypi](#) v1.1.12

[GitHub](#) v1.1.12

# Includes for Foliant

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 `processors` section in the project config:

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

The preprocessor has a number of options:

```
1 preprocessors:
2     - includes:
3         cache_dir: !path .includescache
4         recursive: true
5         extensions:
6             - md
7             - j2
8         aliases:
9             ...
```

**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.

**extensions** List of file extensions that defines the types of files which should be processed looking for include statements. Might be useful if you need to include some content from third-party sources into non-Markdown files like configs, templates, reports, etc. Defaults to `[md]`.

**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:

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

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

```
1 <include>$foo$path/to/doc.md</include>  
2  
3 <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:

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

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

```
1 Text below is taken from a remote repository.  
2  
3 <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:

```
1 Text below is taken from a remote repository on branch  
develop.  
2  
3 <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.

**url** HTTP(S) URL of the content that should be included.

**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, `url` is required to include a remote 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. If the `to_heading`, `to_id`, or `to_end` attribute is not specified, the preprocessor cuts the included content to the next heading of the same level.

**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`. If the `to_heading`, `to_id`, or `to_end` attribute is not specified, the preprocessor cuts the included content to the next heading of the same level.

**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:

```
1 ## Some Heading {#custom_id}
2
3 <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.

**wrap\_code** Attribute that allows to mark up the included content as fence code block or inline code by wrapping the content with additional Markdown syntax constructions. Available values are: `triple_backticks` –to add triple backticks separated with newlines before and after the included content; `triple_tildas`–to do the same but using triple tildas; `single_backticks`–to add single backticks before and after the included content without adding extra newlines. Note that this attribute doesn't affect the included content. So if the content consists of multiple lines, and the `wrap_code` attribute with the value `single_backticks` is set, all newlines within the content will be kept. To perform forced conversion of multiple lines into one, use the `inline` attribute.

**code\_language** Language of the included code snippet that should be additionally marked up as fence code block by using the `wrap_code` attribute with the value `triple_backticks` or `triple_tildas`. Note that the `code_language` attribute doesn't take effect to inline code that is obtained when the `single_backticks` value is used. The value of this attribute should be a string without whitespace characters, usually in lowercase; examples: `python`, `bash`, `json`.

#### 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:

```
1 # Title  
2  
3 ## Subtitle  
4  
5 <include src="other.md" sethead="3"></include>
```

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

```
1 # My Custom Heading  
2
```

```
3 <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 need to include a section with a custom heading:

```
1 # My Custom Heading
2
3 <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:

```
1 Text below is taken from another document.  
2  
3 <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:

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

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

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

And then in the source:

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

You can also specify a particular branch or revision:

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

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

```
1 Include content from “”Intro up to “”Credits:  
2  
3 <include>sample.md#Intro:Credits</include>  
4  
5 Include content from start up to “”Credits:  
6
```

```
7 <include>sample.md#:Credits</include>
8
9 Include content from “”Intro up to the next heading of the
same level:
10
11 <include>sample.md#Intro</include>
```

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

## Macros

[pypi](#) v1.0.4

[GitHub](#) v1.0.4

### Macros for Foliant

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 `processors` and listing your macros in `macros` dictionary:

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

## Usage

Here's the simplest usecase for macros:

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

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

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

Macros support params. This simple feature may make your sources a lot tidier:

```
1 preprocessors:
2   - macros:
3     macros:
4       jira: "https://mycompany.jira.server.us/jira/ticket?
ID={ticket_id}"
```

Now you don't need to remember the address of your Jira server if you want to reference a ticket:

```
Link to jira ticket: <macro ticket_id="DOC-123">jira</macro>
```

## Realworld example

You can combine Macros with tags by other Foliant preprocessors.

This can be 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 the [Flags](#) preprocessor and its `<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:

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

And use it in the source:

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

Just remember, that in this use case `macros` preprocessor must go before `flags` preprocessor in the config. This way macros will be already resolved at the time `flags` starts working.

## Mermaid

[pypi](#) v1.0.1

### 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:

```
1 $ npm install mermaid # installs locally  
2 $ npm install mermaid.cli
```

## Config

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

```
1 preprocessors:  
2     - mermaid
```

The preprocessor has a number of options:

```
1 preprocessors:  
2     - mermaid:  
3         cache_dir: !path .diagramscache  
4         mermaid_path: !path node_modules/.bin/mmdc  
5         format: svg  
6         params:  
7             ...
```

**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:

```
1 preprocessors:  
2     - mermaid:
```

```
3     params:  
4         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:

```
1 ,  
2 Heres a diagram:  
3  
4 <mermaid>  
5 graph TD;  
6     A-->B;  
7 </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.

```
1 Diagram with a caption:  
2  
3 <mermaid caption="Deployment diagram"  
4             params="theme: dark">  
5 </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'}`.

## MetaGraph

pypi v0.1.2

## MetaGraph preprocessor for Foliant

Preprocessor generates Graphviz diagrams of meta sections in the project.

### Installation

```
$ pip install foliantcontrib.metagraph
```

### Config

```
1 preprocessors:
2     - metagraph:
3         natural: false
4         directed: false
5         draw_all: false
```

**natural** if `true` – the graph is generated based on “natural” section structure: main sections are connected to the inner sections, which are connected to their child sections and so on. If `false` – the connections are determined by the `relates` meta section of each chapter. Default: `false`

**directed** If `true` – draws a directed graph (with arrows). Default: `false`

**draw\_all** If `true` – draws all sections, except those which have meta field `draw: false`. If `false` – draws only sections which have meta field `draw: true`. Default: `false`

### Usage

First set up a few meta sections:

```
1 <meta title="Main document" id="main" relates="['first', 'sub']" draw="true"></meta>
2
3 # First title
4 <meta id="first" draw="true"></meta>
5
```

```
6 Lorem ipsum dolor sit amet, consectetur adipisicing elit.  
7 Nesciunt, atque.  
8  
9  
10 <meta id="sub" draw="true"></meta>
```

Then add a `metagraph` tag somewhere in the project:

```
<metagraph></metagraph>
```

## 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 with processing of extremely wide tables with pandoc. Conversion 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 `processors` section in the project config:

```
1 preprocessors:  
2   - multilinetables
```

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

```
1 preprocessors:  
2   - multilinetables:  
3     rewrite_src_files: false  
4     min_table_width: 100  
5     keep_narrow_tables: true  
6     table_columns_to_scale: 3
```

```
7      enable_hyphenation: false
8      hyph_combination: '<br>'
9      convert_to_grid: false
10     targets:
11       - docx
12       - 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 pre-processor applies to all targets.

## Usage

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

## Pgsqldoc

[pypi](#) v1.1.6

[GitHub](#) v1.1.6

## PostgreSQL Docs Generator for Foliant

This preprocessor is DEPRECATED. Please, use [DBDoc](#) instead.

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:

```
1 preprocessors:
2     - pgsqldoc
```

The preprocessor has a number of options:

```
1 preprocessors:
2     - pgsqldoc:
3         host: localhost
4         port: 5432
5         dbname: postgres
6         user: postgres
7         password: ''
8         draw: false
9         filters:
10            ...
11         doc_template: pgsqldoc.j2
12         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`  
**password** 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:

```
1 # Introduction
2
3 This document contains the most awesome automatically
   generated documentation of our marvellous database.
4
5 <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:

```
1 # Introduction
2
3 Introduction text for database documentation.
4
5 <pgsqldoc draw="true"
6       host="11.51.126.8"
7       port="5432"
8       dbname="mydb"
9       user="scott"
10      password="tiger">
11 </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:

```
1 preprocessors:
2   - pgsqldoc:
3     filters:
4       eq:
5         schema: public
6       regex:
7         table_name: 'main_.'
```

or in tag options using the same yaml-syntax:

```
1 <pgsqldoc filters="
2 eq:
3   schema: public
4   regex:
5     table_name: 'main_.'">
6 </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:

```
1 filters:  
2   <operator>:  
3     <field>: value
```

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

```
1 filters:  
2   in:  
3     schema:  
4       - public  
5       - 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>/schema.j2` for database schema 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

[pypi](#) v1.0.6

## PlantUML Diagrams Preprocessor for Foliant

[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 `processors` section in the project config:

```
1 preprocessors:
2   - plantuml
```

The preprocessor has a number of options:

```
1 preprocessors:
2   - plantuml:
3     cache_dir: !path .diagramscache
4     plantuml_path: plantuml
5     params:
6       ...
7     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:

```
1 preprocessors:
2   - plantuml:
3     params:
4       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):

```
1 ,
2 Heres a diagram:
3
4 <plantuml>
5   @startuml
6   ...
7   @enduml
8 </plantuml>
```

To set a caption, use `caption` option:

```
1 Diagram with a caption:
2
3 <plantuml caption="Sample diagram from the official site">
4   @startuml
5   ...
6   @enduml
```

```
7 </plantuml>
```

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

```
1 By default, diagrams are in PNG. But this diagram is in EPS:  
2  
3 <plantuml caption="Vector diagram" format="eps">  
4     @startuml  
5         ...  
6     @enduml  
7 </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.

## RAMLDoc

[pypi v1.0.1](#)

## RAML API Docs Generator for Foliant

This preprocessor generates Markdown documentation from [RAML](#) spec files. It uses `raml2html` converter with [raml2html-full-markdown-theme](#).

`raml2html` uses [Nunjucks](#) templating system.

### Installation

First install `raml2html` and the markdown theme:

```
$ npm install -g raml2html raml2html-full-markdown-theme
```

Then install the preprocessor:

```
$ pip install foliantcontrib.ramldoc
```

## Config

To enable the preprocessor, add `ramldoc` to `processors` section in the project config:

```
1 preprocessors:  
2     - ramldoc
```

The preprocessor has a number of options:

```
1 preprocessors:  
2     - ramldoc:  
3         spec_url: http://localhost/my_api.raml  
4         spec_path: !path my_api.raml  
5         template_dir: !path custom_templates  
6         raml2html_path: raml2html
```

**spec\_url** URL to RAML spec file. If it is a list – preprocessor picks the first working URL.

**spec\_path** Local path to RAML spec file.

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

**template\_dir** Path to directory with [Nunjucks](#) templates. If not specified – default template is used. The main template in the directory must have a name `root.nunjucks`.

**raml2html\_path** Path to raml2html binary. Default: `raml2html`

## Usage

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

```
1 # Introduction  
2  
3 This document contains the automatically generated  
documentation of our API.  
4  
5 <ramldoc></ramldoc>
```

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

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

```
1 # Introduction
2
3 Introduction text for API documentation.
4
5 <ramldoc spec_url="http://localhost/my_api.raml"
6         template_dir="assets/templates">
7 </ramldoc>
```

Tag parameters have the highest priority.

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

## Customizing output

The output markdown is generated by `raml2html` converter, which uses [Nunjucks](#) templating engine (with syntax similar to [Jinja2](#)). If you want to create your own template or modify the default one, specify the `template_dir` parameter.

The main template file in template dir must be named `root.nunjucks`.

You may use the [default template](#) as your starting point.

## RemoveExcess

`RemoveExcess` is a preprocessor that removes unnecessary Markdown files that are not mentioned in the project's `chapters`, from the temporary working directory.

### Installation

```
$ pip install foliantcontrib.removeexcess
```

### Config

To enable the preprocessor, add `removeexcess` to `processors` section in the project config:

```
1 preprocessors:  
2     - removeexcess
```

The preprocessor has no options.

## Usage

By default, all preprocessors are applied to each Markdown source file copied into the temporary working directory.

Often it's needed not to include some files to the project's `chapters`. But anyway, preprocessors will be applied to all source files, that will take extra time and may cause extra errors. Also, extra files may pass to backends that might be undesirable for security reasons.

When RemoveExcess preprocessor is enabled, unnecessary files will be deleted. Decide at your discretion to which place in the preprocessor queue to put it.

# 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 `processors` section in the project config:

```
1 preprocessors:  
2     - replace
```

The preprocessor has two options (default values stated):

```
1 preprocessors:  
2     - replace:  
3         dictionary_filename: replace_dictionary.yml  
4         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:

```
1 words:
2   cod: CoD
3   epg: EPG
4   vod: VoD
5 regexs:
6   '!\\w*!': ''
7   '\\. *(\\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. Applying of the preprocessor to subprojects allows to get links to separate repositories from different pages of a single site (e.g. generated with MkDocs).

By default, the preprocessor emulates MkDocs behavior. The preprocessor generates HTML hyperlink with specific attributes and inserts the link after the first heading of the document. The default behavior may be overridden.

The preprocessor supports the same options `repo_url` and `edit_uri` as MkDocs.

## Installation

RepoLink preprocessor is a part of MultiProject extension:

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

## Usage

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

```
1 preprocessors:  
2     - repolink
```

The preprocessor has a number of options:

```
1 preprocessors:  
2     - repolink:  
3         repo_url: https://github.com/foliant-docs/docs/  
4         edit_uri: /blob/master/src/  
5         link_type: html  
6         link_location: after_first_heading  
7         link_text: "&#xE3C9;"  
8         link_title: View the source file  
9         link_html_attributes: "class=\"md-icon md-content-icon\" style=\"margin: -7.5rem 0\""  
10        targets:  
11            - 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\_type** Link type: HTML (`html`) or Markdown (`markdown`). Default value is `html`.

**link\_location** Place in the document to put the hyperlink. By default, the hyperlink is placed after the first heading, and newlines are added before and after it (`after_first_heading`). Other values: `before_content`—the hyperlink is placed before the content of the document, the newline after it is provided; `after_content`—the hyperlink is placed after the content of the document, the newline before it is added; `defined_by_tag`—the tags `<repo_link></repo_link>` that are met in the content of the document are replaced with the hyperlink.

**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`. This option takes effect only when `link_type` is set to `html`.

**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. This option takes effect only when `link_type` is set to `html`.

**targets** Allowed targets for the preprocessor. If not specified (by default), the pre-processor 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 `runcmds` to `processors` section in the project config, and specify the commands to run:

```
1 preprocessors:
2   - runcmds:
3     commands:
4       - ./build.sh
5       - echo "Hello World" > ${WORKING_DIR}/hello.txt
6     targets:
7       - pre
8       - tex
9       - pdf
10      - docx
```

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

**targets** Allowed targets for the preprocessor. If not specified (by default), the pre-processor 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`.

## ShowCommits

[pypi](#) v1.0.2

[GitHub](#) v1.0.2

### ShowCommits Preprocessor

ShowCommits is a preprocessor that appends the history of Git commits corresponding to the current processed file to its content.

#### Installation

```
$ pip install foliantcontrib.showcommits
```

#### Config

To enable the preprocessor, add `showcommits` to `processors` section in the project config:

```
1 preprocessors:
```

```
2     - showcommits
```

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

```
1 preprocessors:
2     - showcommits:
3         repo_path: !rel_path ./      # Path object that points
4             to the current Foliant 'projects' top-level ("root)
5             directory when the preprocessor initializes
6             try_default_path: true
7             remote_name: origin
8             self-hosted: gitlab
9             protocol: https
10            position: after_content
11            date_format: year_first
12            escape_html: true
13            template: |
14                ## File History
15
16                {{startcommits}}
17                Commit: [{{hash}}]({{url}}), author: [{{author
18 }}]({{email}}), date: {{date}}
19
20                {{message}}
21
22                ``diff
23                {{diff}}
24                ...
25
26                {{endcommits}}
27
28                targets: []
```

**repo\_path** Path to the locally cloned copy of the Git repository that the current Foliant project's files belong to.

**try\_default\_path** Flag that tells the preprocessor to try to use the default `repo_path` if user-specified `repo_path` does not exist.

**remote\_name** Identifier of remote repository; in most cases you don't need to override the default value.

**self-hosted** String that defines the rules of commit's web URL anchor generation when a self-hosted Git repositories management system with web interface is used. Supported values are: `github` for GitHub, `gitlab` for GitLab, and `bitbucket` for BitBucket. If the repo fetch URL hostname is `github.com`, `gitlab.com`, or `bitbucket.org`, the corresponding rules are applied automatically.

**protocol** Web interface URL prefix of a repos management system. Supported values are `https` and `http`.

**position** String that defines where the history of commits should be placed. Supported values are: `after_content` for concatenating the history with the currently processed Markdown file content, and `defined_by_tag` for replacing the tags `<commits></commits>` with the history.

**date\_format** Output date format. If the default value `year_first` is used, the date "December 11, 2019" will be represented as `2019-12-11`. If the `day_first` value is used, this date will be represented as `11.12.2019`.

**escape\_html** Flag that tells the preprocessor to replace HTML control characters with corresponding HTML entities in commit messages and diffs: `&` with `&amp;`; `<` with `<lt;`; `>` with `>gt;`; `"` with `"`.

**template** Template to render the history of commits. If the value is a string that contains one or more newlines (`\n`) or double opening curly braces (`{}`), it is interpreted as a template itself. If the value is a string without newlines and any double opening curly braces, or a `Path` object, it is interpreted as a path to the file that contains a template. Template syntax is described below.

**targets** Allowed targets for the preprocessor. If not specified (by default), the pre-processor applies to all targets.

## Usage

You may override the default template to customize the commits history formatting and rendering. Feel free to use Markdown syntax, HTML, CSS, and JavaScript in your custom templates.

In templates, a number of placeholders is supported.

**`{{startcommits}}`** Beginning of the list of commits that is rendered within a loop. Before this placeholder, you may use some common stuff like an introducing heading or a stylesheet.

**`{{endcommits}}`** End of the list of commits. After this placeholder, you also may use some common stuff like a paragraph of text or a script.

The following placeholders affect only between the `{{startcommits}}` and `{{endcommits}}`.

`{{hash}}` First 8 digits of the commit hash, e.g. `deadc0de`.

`{{url}}` Web URL of commit with an anchor that points to the certain file, e.g. `https://github.com/foliant-docs/foliant/commit/67138f7c#diff-478b1f78b2146021bce46fbf833eb636`. If you don't use a repos management system with web interface, you don't need this placeholder.

`{{author}}` Author name, e.g. Artemy Lomov.

`{{email}}` Author email, e.g. `artemy@lomov.ru`.

`{{date}}` Formatted date, e.g. `2019-12-11`.

`{{message}}` Commit message, e.g. `Bump version to 1.0.1..`

`{{diff}}` Diff between the currently processed Markdown file at a certain commit and the same file at the previous state.

## SuperLinks

[pypi v1.0.4](#)

[GitHub v1.0.4](#)

## SuperLinks for Foliant

This preprocessor extends the functionality of Markdown links, allowing you to reference by the heading title, file name or meta id. It works correctly with most backends, resolving to proper links, depending on which backend is being used.

It adds the `<link>` tag.

### The Problem

The problem with Markdown links is that you have to specify the correct anchor and file path right away.

Let's imagine that you want to create a link to a heading which is defined in another chapter.

If you are building a single-page PDF with Pandoc, you will only need to specify the anchor, which Pandoc generates from that title. But if you are building an MkDocs website, you will need to specify the relative filename to the referenced chapter, and the anchor, which MkDocs generates from the titles. By the way, Pandoc and MkDocs generate anchors differently. So there's no way to make it work for all backends by using just Markdown link syntax.

Superlinks aim to solve this problem.

## Installation

Install the preprocessor:

```
$ pip install foliantcontrib.superlinks
```

## Config

To enable the preprocessor, add `superlink` to `processors` section in the project config.

```
1 preprocessors:  
2     - superlinks
```

**Important:** SuperLinks preprocessor is very sensitive to its position in the preprocessors list. If you are using it in along with `Includes`, `Anchors` or `CustomIDs` preprocessor, the order in which they are mentioned must be following:

```
1 preprocessors:  
2     - includes    # includes must be defined BEFORE  
3         - ...  
4         - superlinks  
5         - ...        # following preprocessors must be defined  
6             - anchors  
7             - customids
```

The preprocessor has no config options just now. But it has some important tag options.

## Usage

To add a link, use a `Link` tag with a combination of following parameters:

**title** Heading title, which you want to create a link to.

**src** Relative path to a chapter which is being referenced.

**meta\_id** ID of the meta section which is being referenced. (if `title` is used, then this title MUST be inside this meta section)

**anchor** Name of the anchor defined by [Anchors](#) preprocessor or an ID defined by [CustomIDs](#) preprocessor. If `src` or `meta` is not provided – the id will be searched globally.

**id** Just a hardcoded id. No magic here.

## Examples

### Reference a title in the same document

```
<link title="My title">Link caption</link>
```

### Reference a title in another chapter

```
<link src="subfolder/another_chapter.md" title="Another title">Link caption</link>
```

### Reference the beginning of another chapter

```
<link src="subfolder/another_chapter.md">Link caption</link>
```

### Reference a title inside meta section

```
<link meta_id="SECTION1" title="Title in section1">Link caption</link>
```

### Reference the beginning of meta section

```
<link meta_id="SECTION1">Link caption</link>
```

### Reference an anchor and search for it globally

```
<link anchor="my_anchor">Link caption</link>
```

### Reference an anchor and search for it in specific chapter

```
<link src="subfolder/another_chapter.md" anchor="my_anchor">  
Link caption</link>
```

Supported Backends:

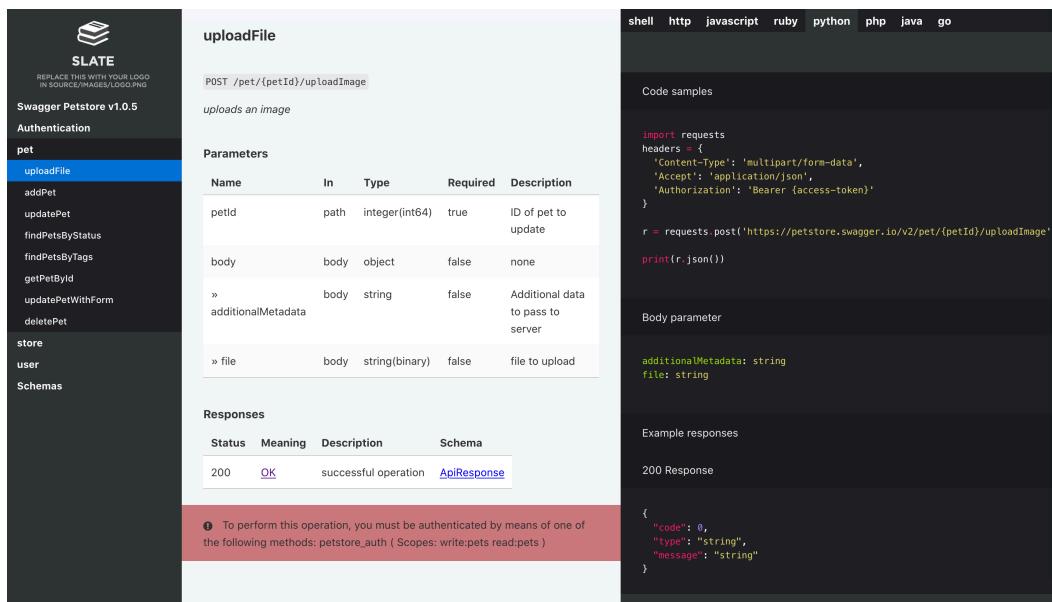
Backend	Support
aglio	□ YES
pandoc	□ YES
mdtopdf	□ YES
mkdocs	□ YES
slate	□ YES
confluence	□ NOT yet

## SwaggerDoc

[pypi](#) v1.2.2

[GitHub](#) v1.2.2

# Swagger API Docs Generator for Foliant



The static site on the picture was built with [Slate](#) backend together with [SwaggerDoc preprocessor](#)

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
```

This preprocessor requires [Widdershins](#) to be installed on your system (unless you are using [Foliant with Full Docker Image](#)):

```
npm install -g widdershins
```

## Config

To enable the preprocessor, add `swaggerdoc` to `processors` section in the project config:

```
1 preprocessors:  
2     - swaggerdoc
```

The preprocessor has a number of options:

```
1 preprocessors:
2   - swaggerdoc:
3     spec_url: http://localhost/swagger.json
4     spec_path: swagger.json
5     additional_json_path: tags.json
6     mode: widdershins
7     template: swagger.j2
8     environment: env.yaml
```

**spec\_url** URL to Swagger spec file. If it is a list – preprocessor takes the first url which works.

**spec\_path** Local path to Swagger spec file (relative to project dir).

If both url and path are specified – preprocessor first tries to fetch spec 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:

```
1 # Introduction
2
3 This document contains the automatically generated
4 documentation of our API.
5 <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:

```
1 # Introduction
2
3 Introduction text for API documentation.
4
5 <swaggerdoc spec_url="http://localhost/swagger.json"
6           mode="jinja"
7           template="swagger.j2">
8 </swaggerdoc>
9
10 <swaggerdoc spec_url="http://localhost/swagger.json"
11        mode="widdershins"
12        environment="env.yml">
13 </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

### 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:

```
1 preprocessors:  
2     - swaggerdoc:  
3         spec_path: swagger.yml  
4         environment:  
5             user_templates: !path ./widdershins_templates/
```

## 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`.

## TemplateParser

[pypi](#) v1.0.4

[GitHub](#) v1.0.4

## 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.

```
1 preprocessors:
2     - templateparser:
3         engine: jinja2
4         engine_params:
5             root: '/usr/src/app'
6         context:
7             param1: 1008
8             param2: 'Kittens'
9         ext_context: context.yml
10        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 transferred 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), or URL to such file on the remote server.

All parameters with other names are also transferred 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:

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

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

```
1 This is ordinary markdown text.  
2  
3 This is a Jinja2 template:  
4 I can count to five!  
5 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:

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

## Sending variables to template

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

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

```
6 {% endfor %}  
7 </jinja2>
```

Result:

```
1 Hi, my name is Andy!  
2 I am 8 years old.  
3  
4 The year before I was 7 years old.  
5  
6 The year before I was 6 years old.  
7  
8 The year before I was 5 years old.  
9  
10 The year before I was 4 years old.  
11  
12 The year before I was 3 years old.  
13  
14 The year before I was 2 years old.  
15  
16 The year before I was 1 years old.
```

Also, you can supply a yaml-file with variables in an `ext_context` parameter:

```
1 <jinja2 ext_context="swagger.yaml">  
2 Swagger file version: {{ swagger }}  
3 Base path: {{ base_path }}  
4 ...  
5 </jinja2>
```

### Pro tip #1

All context variables are also available in the `_foliant_context` dictionary. It may be handy if you don't know at design-time which key names are supplied in the external context file:

```
1 <jinja2 ext_context="customers.yml">  
2 {% for name, data in _foliant_context.items() %}  
3  
4 # Customer {{ name }}
```

```
5
6 Purchase: {{ data['purchase'] }}
7 Order id: {{ data['order_id'] }}
8
9 {% endfor %}
10 </jinja2>
```

### Pro tip #2

If your context file is inside private git repository, you can utilize the power of [Includes](#) preprocessor to retrieve it.

1. Create a file in your `src` dir, for example, `context.md` (`md` extension is obligatory, includes only process markdown files).
2. Add an includes tag:

```
<include      repo_url="https://my_login:my_password@my.git.org/my_repo.git"
path="path/to/file.yml">
```

3. And supply path to this file in your `ext_context` param:

```
<jinja2 ext_context="context.md">
```

### Pro tip #3

If data inside your external context file is not a dictionary, it will be available inside template under `context` variable (or `_foliant_context['context']`).

## Integration with metadata

Templates support latest Foliant [metadata](#) functionality. You can find the meta dictionary for current section under `meta` variable inside template:

```
1 <meta status="ready" title="Custom Title" author="John"></
  meta>
2
3 <jinja2>
4 Document status: {{ meta.status }}
5
6 The document "{{ meta.title }}" is brought to you by {{ meta
  .author }}
7 </jinja2>
```

Result:

```
1 Document status: ready  
2  
3 The document "Custom Title" is brought to you by John
```

You can also find the whole project's Meta object under `meta_object` variable inside template:

```
1 <meta status="ready" title="Custom Title" author="John"></  
  meta>  
2  
3 <jinja2>  
4 List of chapters in this project:  
5 {% for chapter in meta_object.chapters %}  
6 * {{ chapter.name }}  
7 {% endfor %}  
8 </jinja2>
```

Result:

```
1 List of chapters in this project:  
2  
3 * index  
4 * sub  
5 * auth
```

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 `processors` section in the project config. The preprocessor has a number of options (best values are set by default where possible):

```
1 preprocessors:
2   - testrail:
3     testrail_url: http://testrails.url
4       \\ Required
5     testrail_login: username
6       \\ Required
7     testrail_pass: password
8       \\ Required
9     project_id: 35
10      \\ Required
11     suite_ids:
12       \\ Optional
13     section_ids:
14       \\ Optional
15     exclude_suite_ids:
16       \\ Optional
17     exclude_section_ids:
18       \\ Optional
19     exclude_case_ids:
20       \\ Optional
21     filename: test_cases.md
22       \\ Optional
23     rewrite_src_files: false
24       \\ Optional
25     template_folder: case_templates
26       \\ Optional
27     img_folder: testrail_imgs
28       \\ Optional
29     move_imgs_from_text: false
30       \\ Optional
```

```

17     section_header: Testing program
18             \\ Recommended
19     std_table_header: Table with testing results
20             \\ Recommended
21     std_table_column_headers: №; Priority; Platform; ID;
22     Test case name; Result; Comment    \\ Recommended
23     add_std_table: true
24             \\ Optional
25     add_suite_headers: true
26             \\ Optional
27     add_section_headers: true
28             \\ Optional
29     add_case_id_to_case_header: false
30             \\ Optional
31     add_case_id_to_std_table: false
32             \\ Optional
33     multi_param_name:
34             \\ Optional
35     multi_param_select:
36             \\ Optional
37     multi_param_select_type: any
38             \\ Optional
39     add_cases_without_multi_param: true
40             \\ Optional
41     add_multi_param_to_case_header: false
42             \\ Optional
43     add_multi_param_to_std_table: false
44             \\ Optional
45     checkbox_param_name:
46             \\ Optional
47     checkbox_param_select_type: checked
48             \\ Optional
49     choose_priorities:
50             \\ Optional
51     add_priority_to_case_header: false
52             \\ Optional

```

```

35     add_priority_to_std_table: false
36             \\ Optional
37     resolve_urls: true
38             \\ Optional
39     screenshots_url: https://gitlab_repository.url
40             \\ Optional
41     img_ext: .png
42             \\ Optional
43     print_case_structure: true
44             \\ 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](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:

```

1 title: &title Test procedure
2
3 chapters:
4   - intro.md
5   - conditions.md

```

```

6      - awesome_test_cases.md <- This one for test cases
7      - appendum.md
8
9 preprocessors:
10     - testrail:
11         testrail_url: http://testrails.url
12         testrail_login: username
13         testrail_pass: password
14         project_id: 35
15         filename: awesome_test_cases.md

```

**rewrite\_src\_files** 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:

```

1 {% if case['custom_steps_separated'][0]['content'] %}
2 {% if case['custom_preconds'] %}
3 **Preconditions:**
4
5 {{ case['custom_preconds'] }}
6 {% endif %}
7
8 **Scenario:**
9
10 {% for case_step in case['custom_steps_separated'] %}
11
12 *Step {{ loop.index }}.* {{ case_step['content'] }}

```

```

13
14 *Expected result:*
15
16 {{ case_step['expected'] }}
17
18 {% endfor %}
19 {% 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):

```

1 case = {
2     'created_by': 3,
3     'created_on': 1524909903,
4     'custom_expected': None,
5     'custom_goals': None,
6     'custom_mission': None,
7     'custom_preconds': '- The user is not registered in the
system.\r\n'
8         '- Registration form opened.',
9     'custom_steps': '',
10    'custom_steps_separated': [{{
11        'content': 'Enter mobile phone number.',
12        'expected': '- Entered phone number '
13        'is visible in the form field.'
14    },
15        {'content': 'Press OK button.',
16        'expected': '- SMS with registration code '
17        'received.\n'}],
18    'custom_test_androidtv': None,
19    'custom_test_appletv': None,
20    'custom_test_smarttv': 'None',
21    'custom_tp': True,
22    'estimate': None,
23    'estimate_forecast': None,

```

```

24     'id': 15940,
25     'milestone_id': None,
26     'priority_id': 4,
27     'refs': None,
28     'section_id': 3441,
29     'suite_id': 101,
30     'template_id': 7,
31     'title': 'Registration by mobile phone number.',
32     'type_id': 7,
33     'updated_by': 10,
34     'updated_on': 1528978979
35 }
```

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

```

1 params = {
2     'multi_param_name': 'platform',
3     'multi_param_sys_name': 'custom_platform',
4     'multi_param_select': ['android', 'ios'],
5     'multi_param_select_type': any,
6     'add_cases_without_multi_param': False,
7     'checkbox_param_name': 'publish',
8     'checkbox_param_sys_name': 'custom_publish',
9     'checkbox_param_select_type': 'checked',
10    'choose_priorities': ['critical', 'high', 'medium'],
11    'add_multi_param_to_case_header': True,
12    'add_multi_param_to_std_table': True,
13    'add_priority_to_case_header': True,
14    'add_priority_to_std_table': True,
15    'add_case_id_to_case_header': False,
16    'add_case_id_to_std_table': False,
17    'links_to_images': [
18        {'id': '123', 'link': '![[Image caption](testrail_imgs/123.png)]'},
19        ...
20    ]
21 }
```

**img\_folder** Folder to store downloaded images if `rewrite_src_files=True`.  
**move\_imgs\_from\_text** It's impossible to compile test cases with images to the table. So you can use this parameter to convert image links in test cases to ordinary markdown-links and get the list with all image links in `params['links_to_images']` parameter to use in jinja template. In this case you'll have to use `multilinetables` and `anchors` preprocessors.

For example, you have 2-step test case:

```
1 Step 1:  
2  
3 Press the button:  
4  
5 ! [Button](index.php?/attachments/get/740)  
6  
7 Result 1:  
8  
9 Dialog box will opened:  
10  
11 ! [Dialog box](index.php?/attachments/get/741)  
12  
13 Step 2:  
14  
15 Select option:  
16  
17 ! [List of options](index.php?/attachments/get/742)  
18  
19 Result 2:  
20  
21 Option selected:  
22  
23 ! [Result](index.php?/attachments/get/743)
```

Minimal `multilinetables` and `anchors` preprocessor settings in `foliant.yml` should be like this (more about `multilinetables` parameters see in [preprocessor documentation](#)):

```
1     - anchors
```

```

2     - multilinetables:
3         enable_hyphenation: true
4         hyph_combination: brkln
5         convert_to_grid: true

```

After `testrail` preprocessor process this test case, you will have `params['links_to_images']` parameter with list of image links in order of appearance to use in jinja template:

```

1 [
2     {'id': '740', 'link': '![[Button]](testrail_imgs/740.png)
3     '},
4     {'id': '741', 'link': '![[Dialog box]](testrail_imgs/741.
5     png)'},
6     {'id': '742', 'link': '![[List of options]](testrail_imgs
/742.png)'},
7     {'id': '743', 'link': '![[Result]](testrail_imgs/743.png)
8     '}
9 ]

```

Using this jinja template:

```

1 **Testing procedure:**  

2  

3 | # | Test step           | Expected result      | Passed    |  

4 |---|-----|-----|-----|-----|  

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

6 | {{ loop.index }} | {{ case_step['content']|replace("\n", "brkln") }} | {{ case_step['expected']|replace("\n", "brkln") }} | | |  

7 {% endfor %}  

8  

9 {% if params['links_to_images'] %}  

10 *Images:  

11  

12 {% for image in params['links_to_images'] %}  

13 <anchor>{{ image['id'] }}</anchor>

```

```
14  
15 {{ image['link'] }}  
16  
17 {% endfor %}  
18 {% endif %}
```

The markdown result will be:

```
1 **Testing procedure:**  
2  
3 +---+-----+-----+  
  
4 | # | Test step | Expected  
5 | result | Passed | Comment |  
5 +=====+=====+=====+  
  
6 | 1 | Press the button | Dialog box  
7 | will opened: | | |  
7 | | | |  
8 | | [Button](#740) | [Dialog box |  
9 | ](#741) | | |  
9 | | | |  
10 +---+-----+-----+  
  
11 | 2 | Select option: | Option  
12 | selected: | | |  
12 | | | |  
13 | | [List of options](#742) | [Result |  
14 | ](#743) | | |  
14 +---+-----+-----+  
  
15  
16 *Images:  
17  
18 <anchor>740</anchor>
```

```

19
20 ! [Button](testrail_imgs/740.png)
21
22 <anchor>741</anchor>
23
24 ! [Dialog box](testrail_imgs/741.png)
25
26 <anchor>742</anchor>
27
28 ! [List of options](testrail_imgs/742.png)
29
30 <anchor>743</anchor>
31
32 ! [Result](testrail_imgs/743.png)

```

So you can use links in the table to go to the correspondent image.

**Important!** Anchors must differ, so if one image (with the same image id) will appear in several test cases, this image will be downloaded separately for each appearance and renamed with postfix '-1', '-2', etc.

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.

**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 Nº; Priority; Platform; ID; Test case name; Result; Comment in Russian.

**add\_std\_table** You can exclude (false) a testing table from the document.

**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:

```
1 preprocessors:  
2   - testrail:  
3     ...  
4     multi_param_name: platforms  
5     multi_param_select: android, ios  
6     ...
```

**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 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).

```

1 preprocessors:
2   - testrail:
3     ...
4   choose_priorities: critical, high, medium
5   ...

```

**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 a possibility to store screenshots in TestRail test cases, 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:

```
1 images/|
2   smarttv/
3   |   └── screenshot1_smarttv.png
4   |   └── screenshot2_smarttv.png
5   |   └── ... |
6   androidtv/
7   |   └── screenshot1_androidtv.png
8   |   └── screenshot2_androidtv.png
9   |   └── ... |
10  appletv/
11 |   └── screenshot1_appletv.png
12 |   └── screenshot2_appletv.png
13 |   └── ... |
14  web/
15 |   └── screenshot1_web.png
16 |   └── screenshot2_web.png
17 |   └── ... |
18  screenshot1.png |
19  screenshot2.png |
20  ...
```

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/>.

**img\_ext** Screenshots extension. Yes, it must be only one and the same for all screenshots. Also this parameter used to save downloaded images from TestRail.

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”:

```
1 $ foliant bump
2 Version bumped from 1.0.0 to 1.0.1.
```

Bump major version:

```
1 $ foliant bump -v major
2 Version bumped from 1.0.1 to 2.0.0.
```

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

```
1 $ foliant bump --help
2 usage: foliant bump [-h] [-v VERSION_PART] [-p PATH] [-c
3   CONFIG]
4
4 Bump Foliant project version.
5
6 optional arguments:
7   -h, --help            show this help message and exit
8   -v VERSION_PART, --version-part VERSION_PART
9                         Part of the version to bump: major,
10                          minor, patch, prerelease, or build (default: patch).
11   -p PATH, --path PATH  Path to the directory with the
12     config file (default: ".").
11   -c CONFIG, --config CONFIG
```

```
12           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):

```
1 gupload:
2     gdrive_folder_name: Foliant upload
3     gdrive_folder_id:
4     gdoc_title:
5     gdoc_id:
6     convert_file:
7     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 URLs’.
  - Click ‘Save’.
5. Click ‘Download JSON’ on the right side of Client ID to download `client_secrets.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:

```
1 $ foliant gupload docx✓
2 Parsing config✓
3 Applying preprocessor flatten✓
4 Making docx with Pandoc—————
5
6 Result: filename.docx—————✓
7
8 Parsing config
9 Your browser has been opened to visit:
10
11     https://accounts.google.com/o/oauth2/auth?...
12
13 Authentication successful.✓
```

```
14 Uploading 'filename.docx' to Google Drive
```

---

```
15
```

```
16 Result:
```

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

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

With command line authentication procedure differs little bit:

```
1 $ docker-compose run --rm foliant gupload docx✓  
2 Parsing config✓  
3 Applying preprocessor flatten✓  
4 Making docx with Pandoc—————  
5  
6 Result: filename.docx—————✓  
7  
8 Parsing config  
9 Go to the following link in your browser:  
10  
11     https://accounts.google.com/o/oauth2/auth?...  
12  
13 Enter verification code: 4/XgBllTXpxv8kKjsiTxC  
14 Authentication successful.✓  
15 Uploading 'filename.docx' to Google Drive
```

---

```
16
```

```
17 Result:
```

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

`meta generate` command collects metadata from the Foliant project and saves it into a YAML-file.

### 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:

```
1 meta:  
2     filename: meta.yml
```

**filename** name of the YAML-file with generated project metadata.

## 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:

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

Create project from a custom template:

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

You can provide the project name without user prompt:

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

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

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

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

```
1 $ foliant init --help
2 usage: foliant init [-h] [-n NAME] [-t NAME or PATH] [-q]
3
4 Generate new Foliant project.
5
6 optional arguments:
7   -h, --help            show this help message and exit
```

```
8   -n NAME, --name NAME  Name of the Foliant project
9   -t NAME or PATH, --template NAME or PATH
10                      Name of a built-in project template
11                      or path to custom one
11   -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

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

Or:

```
1 $ foliant init -t preprocessor -n "Awesome Preprocessor"✓
2 Generating project
3
4 Project "Awesome Preprocessor" created in awesome-
preprocessor
```

Result:

```
1 $ tree awesome-preprocessor
2 .
3   +-- changelog.md
4   +-- foliant
5     +-- preprocessors
6       +-- awesome-preprocessor.py
7   +-- LICENSE
8   +-- README.md
9   +-- setup.py
10
11 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 `--path` 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`:

```

1 $ foliant subset --help
2 usage: foliant subset [-h] [-p PROJECT_DIR_PATH] [-c CONFIG]
3   [-n] [-d] SUBPATH
4
5 Generate the config file to build the project subset from
6 SUBPATH.
7 positional arguments:
8   SUBPATH           Path to the subset of the Foliant
9 optional arguments:
10  -h, --help        show this help message and exit
11  -p PROJECT_DIR, --path PROJECT_DIR
12                      Path to the Foliant project
13  -c CONFIG, --config CONFIG
14                      Name of config file of the Foliant
15                      project, default 'foliant.yml'
16  -n, --norewrite    Do not rewrite the paths of Markdown
17                      files in the subset partial config
18  -d, --debug        Log all events during build. If not
19                      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:

```

1 $ tree
2 .
3   |
4   foliant.yml
5   src

```

```
5   └── group_1
6       ├── product_1
7           └── feature_1
8               ├── foliant.yml
9               └── index.md
10      └── product_2
11          ├── foliant.yml
12          └── main.md
13  └── group_2
14      ├── foliant.yml
15      └── intro.md
```

The command `foliant subset group_1/product_1/feaute_1` will merge the files `./src/group_1/product_1/feaute_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`:

```
1 title: &title Default Title
2
3 subtitle: &subtitle Default Subtitle
4
5 version: &version 0.0
6
7 backend_config:
8     pandoc:
9         template: !path /somewhere/template.tex
10        reference_docx: !path /somewhere/reference.docx
11        vars:
12            title: *title
13            version: *version
14            subtitle: *subtitle
15            year: 2018
16        params:
```

```
17          pdf_engine: xelatex
```

The content of the partial config file `./src/group_1/product_1/feautre_1/foliant.yml`:

```
1 title: &title Group 1, Product 1, Feature 1
2
3 subtitle: &subtitle Technical Specification
4
5 version: &version 1.0
6
7 chapters:
8     - index.md
9
10 backend_config:
11     pandoc:
12         vars:
13             year: 2019
```

The content of newly generated file `./foliant.yml.subset`:

```
1 title: &title Group 1, Product 1, Feature 1
2 subtitle: &subtitle Technical Specification
3 version: &version 1.0
4 backend_config:
5     pandoc:
6         template: !path /somewhere/template.tex
7         reference_docx: !path /somewhere/reference.docx
8         vars:
9             title: *title
10            version: *version
11            subtitle: *subtitle
12            year: 2019
13            params:
14                pdf_engine: xelatex
15 chapters:
16 - 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

## AltStructure

pypi v0.2.0

GitHub v0.2.0

### AltStructure Extension

AltStructure is a config extension for Foliant to generate alternative chapter structure based on metadata.

It adds an `alt_structure` preprocessor and resolves `!alt_structure` YAML tags in the project config.

#### Installation

```
$ pip install foliantcontrib.alt_structure
```

#### Configuration

##### Config extension

Options for AltStructure are specified in the `alt_structure` section at the root of Foliant config file:

```
1 alt_structure:
2     structure:
3         topic:
4             entity:
5             additional:
6     add_unmatched_to_root: false
7     registry:
8         auth: Аутентификация и авторизация
9         weather: Погода
```

```
10      test_case: Тест кейсы
11      spec: Спецификации
```

**structure** (required) A mapping tree, representing alternative structure.

**add\_unmatched\_to\_root** if `true`, all chapters that weren't matched to structure in metadata will be added to root of the chapter tree. If `false` – they will be ignored. Default: `false`

**registry** A dictionary which defines aliases for chapter tree categories.

## Preprocessor

Preprocessor has just one option:

```
1 preprocessors:
2     - alt_structure:
3         create_subfolders: true
```

**create\_subfolders** If `true`, preprocessor will create a full copy of the working-dir and add it to the beginning of all filepaths in the generated structure. If `false` – preprocessor doesn't do anything. Default: `true`

## Usage

### Step 1

Add `!alt_structure` tag to your chapters in the place where you expect new structure to be generated. It accepts one argument: list of chapters, which will be scanned.

```
1 chapters:
2     - basic: # <-- this is _chapter tree category_
3         - auth/auth.md
4         - index.md
5         - auth/test_auth.md
6         - auth/test_auth_aux.md
7         - weather.md
8         - glossary.md
9         - auth/spec_auth.md
10        - test_weather.md
11        - Alternative: !alt_structure
```

```
12      - auth/auth.md
13      - index.md
14      - auth/test_auth.md
15      - auth/test_auth_aux.md
16      - weather.md
17      - glossary.md
18      - auth/spec_auth.md
19      - test_weather.md
```

AltStructure extension introduces a lot of new notions, so let's agree on some terms to make sure we are on the same page. [Chapter tree category](#) is a mapping with single key which you add to your chapter list to create hierarchy. `basic:` and `Alternative:` are categories in this example.

You can also utilize YAML anchors and aliases, but in this case, because of language limitation you need to supply alias inside a list. Let's use it to get the same result as the above, but in a more compact way:

```
1 chapters:
2     - basic: &basic
3         - auth/auth.md
4         - index.md
5         - auth/test_auth.md
6         - auth/test_auth_aux.md
7         - weather.md
8         - glossary.md
9         - auth/spec_auth.md
10        - test_weather.md
11    - Alternative: !alt_structure [*basic]
```

## Step 2

Next you need to define the structure in `structure` parameter of extension config. It is defined by a mapping tree of [node types](#). For example:

```
1 alt_structure:
2     structure:
3         topic: # topic is the upmost node type
```

```
4           entity: # nodes with type "entity" will be
5             nested in "topic"
6             additional:
7               glossaries:
```

These names of the node types are arbitrary, you can use any words you like except `root` and `subfolder`.

### Step 3

Open your source md-files and edit their [main meta sections](#). Main meta section is a section, defined before the first heading in the document (check [metadata documentation](#) for more info). Add a mapping with nodes for this chapter under the key `structure`.

file `auth_spec.md`

```
1 ---
2 structure:
3   topic: auth # <-- node type: node name
4   entity: spec
5 ---
6
7 # Specification for authorization
```

Here `topic` and `entity` are node types, which are part of our structure (step 2). `auth` and `spec` are [node names](#). After applying `!alt_structure` tag nodes will be converted into chapter tree categories. Node type defines the level of the category and node name defines the caption of the category.

We've added two nodes to the `structure` field of chapter metadata: `topic: auth` and `entity: spec`. In the structure that we've defined on step 2 the `topic` goes first and `entity` – second. So after applying the tag, this chapter will appear in config like this:

```
1 - auth:
2   - spec:
3     - auth_spec.md
```

If we'd stated only `topic` key in metadata, then it would look like this:

```
1 - auth:
```

```
2     - auth_spec.md
```

#### Step 4

Now let's fill up registry. We used `spec` and `auth` in our metadata for node names, but these words don't look pretty in the documents. Registry allows us to set verbose captions for node names in config:

```
1 alt_structure:
2     structure:
3         - ['topic', 'entity']
4         - 'additional/glossaries'
5     registry:
6         auth: Authentication and Authorization
7         spec: Specifications
```

With such registry now our new structure will look like this:

```
1 - Authentication and Authorization:
2     - Specifications:
3         - auth_spec.md
```

#### Special node types

In the step 2 of the user guide above we've mentioned that you may choose any node names in the structure except `root` and `subfolder`. These are special note types and here's how you can use them.

#### root

For example, if our structure looks like this:

```
1 alt_structure:
2     structure:
3         topic:
4             entity:
```

and our chapter's metadata looks like this:

```
1 ---
2 structure:
3     foo: bar
```

```
4 ---
```

The node `foo: bar` is not part of the structure, so applying the `!alt_structure` tag it will just be ignored (unless `add_unmatched_to_root` is set to `true` in config). But what if you want to add it to the root of your chapter tree?

To do that – add the `root` node to your metadata:

```
1 ---
```

```
2 structure:
3     foo: bar
4     root: true # the value of the key `root` is ignored, we
5     use `true` for clarity
```

```
5 ---
```

### subfolder

By defining `subfolder` node in chapter's metadata you can manually add another chapter tree category to any chapter.

For example:

file auth\_spec.md

```
1 ---
```

```
2 structure:
3     topic: auth
4     entity: spec
5     subfolder: Main specifications
```

```
6 ---
```

After applying tag the new structure will look like this:

```
1 - auth:
2     - spec:
3         - Main specifications:
4             - auth_spec.md
```

### Using preprocessor

By default the `!alt_structure` tag only affects the `chapters` section of your `foliant.yml`. This may lead to situation when the same file is mentioned several times

in the `chapters` section. While most backends are fine with that – they will just publish the file two times, [MkDocs](#) does not handle this situation well.

That's where you will need to add the preprocessor `alt_structure` to your preprocessors list. Preprocessor creates a subfolder in the `working_dir` and copies the entire `working_dir` contents into it. Then it inserts the subfolder name into the beginning of all chapters paths in the alternative structure.

**Important:** It is recommended to add this preprocessor to the end of the preprocessors list.

```
1 preprocessors:  
2     ...  
3     alt_structure:  
4         create_subfolders: true
```

Note, that the parameter `create_subfolders` is not necessary, it is `true` by default. But we recommend to state it anyway for clarity.

After applying the tag, your new structure will now look like this:

```
1 - Authentication and Authorization:  
2     - Specifications:  
3         - alt1/auth_spec.md
```

The contents of the `working_dir` were copied into a subdir `alt1`, and new structure refers to the files in this subdir.

## MultiProject

This extension resolves the `!from` YAML tag in the project config and replaces the value of the tag with `chapters` section of related subproject.

Nested subprojects are processed recursively.

## 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:

```
1 chapters:
2   - index.md
3   - !from local_dir
4   - !from https://github.com/foliant-docs/docs.git
5   - !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`).

Limitations:

- directory names of subprojects of the same level should be unique;
- source directories of the multiproject and of all the subprojects should have the same names; also they should be located inside the “root” directories of corresponding projects;
- config files of the multiproject and of all the subprojects should have the same names;
- subprojects from remote Git repositories do not need to be newly cloned before each build, but local subprojects are copied into cache before each build;
- it’s undesirable if the path of the “root” directory of the top-level project contains `.multiprojectcache` directory as its some part.

## 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 `-` delimiter. 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 `.` delimiter.

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:

```
1 title: &title My Awesome Project
2 version: &version 1.0
3 slug: !slug
4     - *title
5     - *version
6     - !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:

```
1 backend_config:
2     pandoc:
3         slug: !slug
4             - *title
5             - *version
6             - !date
7     mkdocs:
8         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:

```
1 version: !version
2     - 1
3     - !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:

```
1 version: !version
2     - 2
3     - !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](#).

## [2020-07-29] `foliantcontrib.includes` 1.1.12

- Add the `wrap_code` and `code_language` attributes to mark up the included content as fence code block or inline code.
- Prevent to create cache directory when it's not needed. Improve code style. Refactor a little.

## [2020-07-22] `foliant` 1.0.12

- Add the `disable_implicit_unescape` option. Remove warning when `escape_code` is not set.
- Support the `!env` YAML tag to use environment variables in the project config.
- Allow to specify custom directory to store logs with the `--logs|-l` command line option.
- Flush output to STDOUT in progress status messages and in the `foliant.utils.output()` method.
- Get and log the names and versions of all installed Foliant-related packages.
- Do not raise exception of the same type that is raised by a preprocessor, raise `RuntimeError` instead because some exceptions take more arguments than one.

## [2020-07-22] `foliantcontrib.pandoc` 1.0.11

- Do not re-raise an exception of the same type as raised, raise `RuntimeError` instead, it's needed to avoid non-informative error messages.

## [2020-07-22] `foliantcontrib.mkdocs` 1.0.12

- Do not re-raise an exception of the same type as raised, raise `RuntimeError` instead, it's needed to avoid non-informative error messages.

## [2020-07-20] [foliantcontrib.multiproject](#) 1.0.14

- Support Foliant Core 1.0.12, write logs to the directory that is specified for the multiproject.

## [2020-07-19] [foliantcontrib.escapecode](#) 1.0.3

- Do not fail the preprocessor if saved code is not found, show warning message instead.

## [2020-07-14] [foliantcontrib.confluence](#) 0.6.12

- New: option to store passwords in passfile.
- New: nohead option to crop first title from the page.
- Fix: better error reporting after updated atlassian-python-api package.
- New: if you specified only `space_key` param in metadata and no `title`, section heading will be used as title.
- Fix: if hierarchy is created on the test run, missing parents by title are now ignored

## [2020-07-09] [foliantcontrib.multilinetables](#) 1.2.3

- Problem with strings containing only hyphens fixed (critical for narrow columns with lists in grid tables).

## [2020-07-09] [foliantcontrib.includes](#) 1.1.11

- Add the `extensions` config parameter to process file types different from `.md`.
- Add the `url` attribute to include content that is available by HTTP(S) URL.

## [2020-06-16] [foliantcontrib.confluence](#) 0.6.11

- Fix: XML error in code block conversion.

## [2020-06-10] [foliantcontrib.testrail](#) 1.3.1

- Now it's possible to use one image in several test-cases and process it correctly with `move_imgs_from_text` parameter.

## [2020-06-09] [foliantcontrib.testrail](#) 1.3.0

- New parameter:
  - move\_imgs\_from\_text – converts image links in test cases to ordinary markdown-links, and adds all links to `params` variable to use in jinja template.
- Some `readme.md` bugs fixed.

## [2020-06-09] [foliantcontrib.flatten](#) 1.0.7

- Fix: bug in rewrite local links regex.

## [2020-06-08] [foliantcontrib.testrail](#) 1.2.2

- Processing of several images in one case-step fixed.

## [2020-06-05] [foliantcontrib.dbdoc](#) 0.1.2

- Fix: schema filter in Oracle functions query

## [2020-06-05] [foliantcontrib.swaggerdoc](#) 1.2.2

- Fix spec path issue.
- Fix: jinja mode default template wasn't copied.

## [2020-06-05] [foliantcontrib.dbdoc](#) 0.1.1

- New: Add views query to components
- Fix: Oracle triggers query
- Fix: Fix both PostgreSQL and Oracle templates

## [2020-06-03] [foliantcontrib.dbmldoc](#) 0.2.4

- Pydbml parser version updated to 0.3.2
- Updated templates

## [2020-06-02] [foliantcontrib.pgsqldoc](#) 1.1.6

- Preprocessor is now deprecated. Please, use DBDoc instead:  
<https://github.com/foliant-docs/foliantcontrib.dbdoc>

## [2020-06-02] [foliantcontrib.dbdoc](#) 0.1.0

- Initial release

## [2020-05-28] [foliantcontrib.testrail](#) 1.2.1

- Bug with copying nonexistent folder to source fixed.

## [2020-05-27] [foliantcontrib.bindfigma](#) 1.0.3

- Fix bug in caching.

## [2020-05-27] [foliantcontrib.bindfigma](#) 1.0.2

- Add `api_caching` option. Add source Markdown file path to the messages written to STDOUT.

## [2020-05-26] [foliantcontrib.testrail](#) 1.2.0

- Downloading of images from test cases implemented.
- New parameter:
  - `img_folder` – folder name to store downloaded images.
- Renamed parameters:
  - `rewrite_src_file` -> `rewrite_src_files`,
  - `screenshots_ext` -> `img_ext`.
- Paths processing fixed.

## [2020-05-22] [foliantcontrib.templateparser](#) 1.0.4

- All variables, supplied in context, are also available inside the `_foliant_context` variable
- You can now supply a link to file on remote server in the `ext_context` parameter.
- External context yaml-file now may be not a dictionary. In this case it will be available under the `context` template variable.

## [2020-05-20] [foliantcontrib.superlinks](#) 1.0.4

- Fix: bug with chapters

## [2020-04-29] [foliantcontrib.dbmldoc](#) 0.2.2

- Pydbml parser version updated to 0.3.2
- Updated templates

## [2020-04-23] [foliantcontrib.archeme](#) 1.0.2

- Fix the same bug in stronger way.

## [2020-04-23] [foliantcontrib.archeme](#) 1.0.1

- Fix very strange bug with modules cache.

## [2020-04-22] [foliantcontrib.dbmldoc](#) 0.1

- Initial release

## [2020-04-17] [foliantcontrib.multiproject](#) 1.0.13

- Keep temporary directories of built subprojects. It is needed when local includes that rewrite image paths are used.

## [2020-04-14] [foliantcontrib.elasticsearch](#) 1.0.4

- Add copy action.

## [2020-04-10] [foliantcontrib.replace](#) 1.0.4

- Replace in links and images fixed.

## [2020-04-10] [foliantcontrib.alt\\_structure](#) 0.2.0

- Preprocessor now doesn't read config file, which previously caused MultiProject to run second time.
- Registry is now flat dictionary.
- Structure is now supplied via dictionary.

## [2020-04-10] [foliantcontrib.showcommits](#) 1.0.2

- Add `try_default_path` and `escape_html` options.

## [2020-04-09] [foliantcontrib.elasticsearch](#) 1.0.3

- Add `require_env` option.

## [2020-04-06] [foliantcontrib.meta](#) 1.3.2

- Cutomids are now cut out from titles.
- Added logging.
- Meta commands now support `--debug -d` and `--quiet -q` arguments.
- `meta generate` command now gives some verbose output after work.
- Fix: `get_section_by_offset` didn't count YFM.

## [2020-04-02] [foliantcontrib.confluence](#) 0.6.10

- Disabled tabbed code blocks conversion because of conflicts.

## [2020-04-02] [foliantcontrib.utils.preprocessor\\_ext](#) 1.0.3

- Add `_process_all_files` method for preprocessors without tags.

## [2020-04-01] [foliantcontrib.testcoverage](#) 0.1.1

- Support meta 1.3.0

## [2020-04-01] [foliantcontrib.testcoverage](#) 0.1.0

- Initial release.

## [2020-04-01] [foliantcontrib.metagraph](#) 0.1.2

- Metadata is now taken from `src_dir` to minimize possible conflicts with other preprocessors.

## [2020-03-27] [foliantcontrib.metagraph](#) 0.1.1

- New parameter: `draw_all`, which controls which sections are included.

## [2020-03-26] [foliantcontrib.metagraph](#) 0.1.0

- Initial release

## [2020-03-26] [foliantcontrib.templateparser](#) 1.0.3

- Now meta dictionary is available inside templates under `meta` variable.
- Project's meta object is available inside templates under `meta_object` variable.

## [2020-03-26] [foliantcontrib.meta](#) 1.3.1

- `remove_meta` now also trims whitespaces in the beginning of the file after removing YFM
- Main section's title is now set to first heading, if:
  - the first heading is a 1-level heading (#),
  - the first heading doesn't have meta.
- Fix: YFM was not included in meta in some cases

## [2020-03-25] [foliantcontrib.confluence](#) 0.6.8

- Now foliant-anchors are always added even for new pages

## [2020-03-25] [foliantcontrib.confluence](#) 0.6.9

- Introducing import from confluence into Foliant with `confluence` tag
- Fix: solved conflicts between inline comments and macros (including anchors)
- Fix: backend crashed if new page content was empty
- Markdown code blocks are now converted into code-block macros
- Markdown task lists are now converted into task-list macros
- New `test_run` option

## [2020-03-23] [foliantcontrib.utils.combined\\_options](#) 1.0.10

- Fix: default dict was overriden after the first use.
- Allow to supply the list of priorities instead of just one priority.
- Priority for those option dictionaries, which are not mentioned in the `priority` param, are now defined by the order dictionaries are defined.

## [2020-03-12] [foliantcontrib.testrail](#) 1.1.11

- Misprint fixed.

## [2020-03-12] [foliantcontrib.testrail](#) 1.1.10

- Bug with template handling fixed.

## [2020-03-11] [foliantcontrib.confluence](#) 0.6.7

- Fix another conflict with `escapecode`

## [2020-03-05] [foliantcontrib.graphviz](#) 1.1.3

- Fix: `as_image` takes effect only with `svg` format.

## [2020-02-28] [foliantcontrib.bindfigma](#) 1.0.1

- Add `hyperlinks` and `multi_delimeter` options.
- Output error messages to STDOUT.
- Minor improvements.

## [2020-02-12] [foliantcontrib.alt\\_structure](#) 0.1.2

- Fix: Remove config check from init

## [2020-02-10] [foliantcontrib.alt\\_structure](#) 0.1.1

- Initial release

## [2020-02-07] [foliantcontrib.utils.combined\\_options](#) 1.0.9

- defaults now actually supply default value (before they were only used for validation)
- add None to possible type validation in `val_type`.

## [2020-02-06] [foliantcontrib.mkdocs](#) 1.0.11

- Get captions for pages from `workingdir` instead of `src_dir`

## [2020-02-04] [foliantcontrib.slate](#) 1.0.8

- Support meta 1.3

## [2020-02-04] [foliantcontrib.superlinks](#) 1.0.3

- Support meta 1.3

## [2020-02-04] [foliantcontrib.includes](#) 1.1.9

- Support meta 1.3.

## [2020-02-04] [foliantcontrib.meta](#) 1.3.0

- Restructure modules to aid import errors. Meta-related functions and classes are now available independantly from `foliant.meta` package.

## [2020-02-04] [foliantcontrib.confluence](#) 0.6.6

- Support meta 1.3
- Now foliant-anchors are always added around uploaded content
- Anchors are now case insensitive

## [2020-02-03] [foliantcontrib.meta](#) 1.2.3

- Add `get_chapter` method to `Meta` class.
- Add Developer's guide to `readme`.

## [2020-01-31] [foliantcontrib.elasticsearch](#) 1.0.2

- Add `format` option. Use `escape_html` only for `format: plaintext`.

## [2020-01-31] [foliantcontrib.elasticsearch](#) 1.0.1

- Add `escape_html` option. Perform actions `delete`, `create` by default. Fix HTML markup in Web application example.

## [2020-01-22] [foliantcontrib.confluence](#) 0.6.5

- Fix: build crashed when several resolved inline comments referred to same string

## [2019-12-24] [foliantcontrib.superlinks](#) 1.0.2

- add dependencies order check
- rename anchor parameter to `id`.
- add anchor parameter for possibly global anchor search.
- link to anchors in Confluence are now partly supported.

## [2019-12-24] [foliantcontrib.utils.header\\_anchors](#) 1.0.1

- Add `is_flat` function to determine flat backends.

## [2019-12-24] [foliantcontrib.anchors](#) 1.0.4

- Applied anchors are now checked from all chapters for flat backends.

## [2019-12-24] [foliantcontrib.testrail](#) 1.1.9

- Function to get case data by id added.

## [2019-12-23] [foliantcontrib.superlinks](#) 1.0.1

- Initial release

## [2019-12-20] [foliantcontrib.utils.header\\_anchors](#) 1.0.0

- Initial release.

## [2019-12-20] [foliantcontrib.anchors](#) 1.0.3

- Better regex patterns.
- Conflicts are now determined for each backend separately.
- Add confluence anchors.

## [2019-12-20] [foliantcontrib.meta](#) 1.2.2

- Don't require empty line between heading and meta tag.
- Allow comments in YFM.
- Better patterns for sections detection.

## [2019-12-12] [foliantcontrib.showcommits](#) 1.0.1

- Fix template processing. Log repo path.

## [2019-12-12] [foliantcontrib.flatten](#) 1.0.6

- Rewrite local links (e.g. `some_file.md#some_id` → `#some_id`).

## [2019-12-04] [foliantcontrib.slate](#) 1.0.7

- Fix: images are preserved in the output, even from subfolders.
- YAML Front Matter from the sources is now ignored.

## [2019-12-02] [foliantcontrib.init](#) 1.0.8

- Add comment to Dockerfile with option to use Foliant full image.
- Remove slugs from docker-compose. Now the service is always named `foliant`.

## [2019-11-22] [foliantcontrib.meta](#) 1.2.1

- Fix bug with imports.

## [2019-11-21] [foliantcontrib.meta](#) 1.2.0

- Support sections
- meta.yml format restructure

## [2019-11-21] [foliantcontrib.confluence](#) 0.6.4

- Support meta 1.2. Now you can publish sections to confluence.

## [2019-11-21] [foliantcontrib.includes](#) 1.1.8

- Support meta 1.2.

## [2019-11-20] [foliantcontrib.imgcaptions](#) 1.0.2

- Fix: `stylesheet_path` only worked with the `!project_path` modifier.
- Add the `template` parameter to customize the caption HTML tag.

## [2019-11-09] [foliantcontrib.mdtopdf](#) 1.0.0

- Initial release.

## [2019-11-06] [foliantcontrib.ramldoc](#) 1.0.1

- Initial release

## [2019-10-28] [foliantcontrib.aglio](#) 1.0.0

- Initial release

## [2019-10-25] [foliantcontrib.slate](#) 1.0.6

- Fix bug with error catching introduced in 1.0.5

## [2019-10-25] [foliantcontrib.slate](#) 1.0.5

- Better error reporting.
- Fixes for working with includes.

## [2019-10-16] [foliantcontrib.escapecode](#) 1.0.2

- Improve flexibility: add new actions, allow to override defaults.

## [2019-10-16] [foliantcontrib.multiproject](#) 1.0.12

- Take into account the `quiet` flag. Require Foliant 1.0.11 for this reason.

## [2019-10-16] [foliantcontrib.includes](#) 1.1.7

- Allow to specify custom options for `EscapeCode` preprocessor as the `escape_code.options` config parameter value.

## [2019-10-16] [foliant](#) 1.0.11

- Allow to specify custom options for `EscapeCode` preprocessor as the `escape_code.options` config parameter value.
- Pass the `quiet` flag to `BaseParser()` as an optional argument for using in config extensions.

## [2019-10-15] [foliantcontrib.subset](#) 1.0.9

- Fix incompatibilities with newer versions of modules: Ciliar, PyYAML.

## [2019-10-10] [foliantcontrib.multiproject](#) 1.0.11

- Allow recursive processing of nested subprojects.
- Allow to specify type (HTML/Markdown) and location for repo links.
- Fix incompatibility with new Ciliar: key names should not contain hyphens.

## [2019-10-07] [foliantcontrib.confluence](#) 0.6.3

- Remove resolved inline comments as they mix up with unresolved.

## [2019-10-04] [foliantcontrib.mermaid](#) 1.0.1

- Better error reporting

## [2019-10-02] [foliantcontrib.utils.combined\\_options](#) 1.0.8

- Fix `validate_exists` validator.

## [2019-10-01] [foliantcontrib.confluence](#) 0.6.2

- Added `parent_title` parameter.
- Fix: images were not uploaded for new pages.

## [2019-10-01] [foliantcontrib.multiproject](#) 1.0.10

- Allow the first heading to be located not in the beginning of a document.

## [2019-09-26] [foliantcontrib.flatten](#) 1.0.5

- Add the `keep_sources` option to keep original files in the temporary working directory after flattening.

## [2019-09-25] [foliantcontrib.confluence](#) 0.6.0

- Now content is put in place of `foliant` anchor or instead of `foliant_start` ... `foliant_end` anchors on the target page. If no anchors on page – content replaces the whole body.
- New modes (backwards compatibility is broken!).
- Now following files are available for debug in cache dir: 1. markdown before conversion to html. 2. Converted to HTML. 3. Final XHTML source which is uploaded to confluence.
- Working (but far from perfect) detection if file was changed.
- Only upload changed attachments.
- Updating attachments instead of deleting and uploading again.

## [2019-09-19] [foliantcontrib.confluence](#) 0.5.2

- Completely rewrite restoring inline comments feature.
- Add `restore_comments` and `resolve_if_changed` emergency options.
- Allow insert raw confluence code (macros, etc) inside `<raw_confluence>` tag.

## [2019-09-19] [foliantcontrib.history](#) 1.0.8

- Allow to ignore merge commits in `from: commits` mode.

## [2019-09-18] [foliantcontrib.history](#) 1.0.7

- Allow to get repo names from README files.

## [2019-09-16] [foliantcontrib.history](#) 1.0.6

- Fix some regex patterns.

## [2019-09-16] [foliantcontrib.history](#) 1.0.5

- Allow to generate history based on tags and commits.

## [2019-09-13] [foliantcontrib.history](#) 1.0.4

- Add templates for target Markdown headings and RSS items titles.

## [2019-09-13] [foliantcontrib.history](#) 1.0.3

- Escape regex metacharacters in headings.

## [2019-09-10] [foliantcontrib.epsconvert](#) 1.0.7

- Fix image reference detection pattern, other minor fixes.

## [2019-09-09] [foliantcontrib.history](#) 1.0.2

- Do not generate common top-level heading of target Markdown content.

## [2019-09-06] [foliantcontrib.history](#) 1.0.1

- Add RSS feed generation.

## [2019-08-28] [foliantcontrib.includes](#) 1.1.6

- Escape regular expression metacharacters in starting and ending headings, IDs, modifiers.

## [2019-08-27] [foliantcontrib.includes](#) 1.1.5

- Remove meta blocks from the included content.

## [2019-08-26] [foliantcontrib.mkdocs](#) 1.0.10

- Fix pattern for heading detection.

## [2019-08-26] [foliantcontrib.swaggerdoc](#) 1.2.0

- Add `spec_path` and `spec_url` parameters.
- All path tag parameters are now loaded relative to current file.
- Better logging and error reporting

## [2019-08-26] [foliantcontrib.utils.combined\\_options](#) 1.0.7

- Add `validate_exists` validator.
- Add `rel_path_convertor`.

## [2019-08-26] [foliantcontrib.customids](#) 1.0.6

- Allow to define custom styles for headings of each level.

## [2019-08-26] [foliantcontrib.confluence](#) 0.4.1

- Fix: conflict with `escape_code`

## [2019-08-23] [foliantcontrib.includes](#) 1.1.4

- Allow for the starting and ending headings to be 1-character long.

## [2019-08-23] [foliantcontrib.confluence](#) 0.4.0

- Fix: attachments were not uploaded for nonexistent pages
- Change confluence api wrapper to `atlassian-python-api`
- Rename backend to `confluence`
- Better error reporting

## [2019-08-23] [foliantcontrib.mkdocs](#) 1.0.9

- Allow the first heading to be located not in the beginning of a document.

## [2019-08-23] [foliantcontrib.epsconvert](#) 1.0.6

- Bug fix: update current directory path before processing of Markdown file content, not after.

## [2019-08-22] [foliantcontrib.imagemagick](#) 1.0.2

- Bug fix: update current directory path before processing of Markdown file content, not after.

## [2019-08-22] [foliantcontrib.meta](#) 1.1.0

- Remove the sections entity.
- Restructure code.

## [2019-08-22] [foliantcontrib.confluence](#) 0.3.0

- Fix bug with images.
- Add multiple modes and mode parameter.
- Add toc parameter to automatically insert toc.
- Fix: upload attachments before text update (this caused images to disappear after manually editing).

## [2019-08-20] [foliantcontrib.meta](#) 1.0.3

- Add span to meta

## [2019-08-16] [foliantcontrib.confluence](#) 0.2.0

- Allow to input login and/or password during build
- Added `pandoc_path` option
- Better logging and error catching

## [2019-08-16] [foliantcontrib.utils.combined\\_options](#) 1.0.6

- Add check for required params.
- Add `val_type` validator to check for param types.
- Allow to set values in Options objects

## [2019-08-15] [foliantcontrib.confluence](#) 0.1.0

- Initial release.

## [2019-08-14] [foliantcontrib.includes](#) 1.1.3

- Allow to specify IDs of anchors in the `from_id` and `to_id` attributes. Support the `to_end` attribute.

## [2019-08-02] [foliantcontrib.escapecode](#) 1.0.1

- Do not ignore diagram definitions. It should be possible to escape the tags used by diagram drawing preprocessors. If some preprocessors need to work with the content that is recognized as code, call `UnescapeCode` explicitly before them.

## [2019-08-01] [foliantcontrib.replace](#) 1.0.3

- Fixed issue with PyYAML deprecated loader.

## [2019-08-01] [foliantcontrib.mermaid](#) 1.0.0

- Initial release

## [2019-07-30] [foliantcontrib.includes](#) 1.1.2

- Fix include statement regex pattern. Tags joined with `|` must be in non-capturing parentheses.

## [2019-07-30] [foliant](#) 1.0.10

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

## [2019-07-30] [foliantcontrib.includes](#) 1.1.1

- Support `escape_code` config option. Require Foliant 1.0.10 and `escapecode` preprocessor 1.0.0.
- Process `sethead` recursively.

## [2019-07-29] [foliantcontrib.utils.preprocessor\\_ext](#) 1.0.2

- Fix output
- Remove `get_options` overriding, it's now implemented similarly in core

## [2019-07-16] [foliantcontrib.bindsympli](#) 1.0.14

- Add `width` attribute to `<sympli>` tag.
- Refactor a little.

## [2019-07-15] [foliantcontrib.slugs](#) 1.0.1

- Add `!version` and `!commit_count` YAML tags.

## [2019-07-09] [foliantcontrib.docus](#) 0.2.0

- More flexible chapters parsing. Lists are now not mandatory.

## [2019-07-09] [foliantcontrib.docus](#) 0.1.0

- Initial release.

## [2019-07-05] [foliantcontrib.runcommands](#) 1.0.1

- Capture the output of an external command and write it to STDOUT.

## [2019-07-01] [foliantcontrib.meta](#) 1.0.2

- Fix: subsections title may be specified in YFM;
- Fix: in subsections title was being cropped out

## [2019-07-01] [foliantcontrib.project\\_graph](#) 1.0.1

- Rename rel attributes: `rel_path` to `path`, `rel_id` to `id`
- Relation types now don't implicitly go to edge labels. Add label explicitly from now on.
- Fixed: relations to `!project_path` and `!rel_path` didn't work.

## [2019-07-01] [foliantcontrib.meta](#) 1.0.1

- Fix: seeds for main sections were not processed.
- Add debug messages for seeds processing.

## [2019-06-28] [foliantcontrib.project\\_graph](#) 1.0.0

Initial release.

## [2019-06-28] [foliantcontrib.meta](#) 1.0.0

Initial release.

## [2019-06-28] [foliantcontrib.includes](#) 1.1.0

- Support Foliant 1.0.9. Add processing of `!path`, `!project_path`, and `!rel_path` modifiers (i.e. YAML tags) in attribute values of pseudo-XML tags inside the included content. Replace the values that preceded by these modifiers with absolute paths resolved depending on current context.
- Allow to specify the top-level (“root”) directory of Foliant project that the included file belongs to, with optional `project_root` attribute of the `<include>` tag. This can be necessary to resolve the `!path` and the `!project_path` modifiers in the included content correctly.
- Allow to specify all necessary parameters of each include statement as attribute values of pseudo-XML tags. Keep legacy syntax for backward compatibility.
- Update README.

## [2019-06-17] [foliant](#) 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.

## [2019-06-14] [foliantcontrib.utils.preprocessor\\_ext](#) 1.0.1

- Support PyYAML 5.1

## [2019-06-14] [foliantcontrib.utils.combined\\_options](#) 1.0.5

- support PyYAML 5.1

## [2019-06-14] [foliantcontrib.templateparser](#) 1.0.2

- support PyYAML 5.1

## [2019-06-14] [foliantcontrib.bindsympli](#) 1.0.13

- Set 2-minutes timeout instead of default 30-seconds when launching Chromium.
- Use `page.waitForSelector()` instead of `page.waitForNavigation()`.
- Use custom `sleep()` function for intentional delays.

## [2019-06-13] [foliantcontrib.badges](#) 1.0.2

- support params which alter badge look to be supplied in tag params

## [2019-06-11] [foliantcontrib.badges](#) 1.0.1

- force img mode on pdf and docx
- add target parameter

## [2019-06-11] [foliantcontrib.badges](#) 1.0.0

- Initial release

## [2019-06-10] [foliantcontrib.admonitions](#) 1.0.0

- Initial release.

## [2019-05-20] [foliantcontrib.graphviz](#) 1.1.1

- Remove src param. (Use includes instead)
- Allow separate tags fail. Preprocessor would issue warning and continue work.

## [2019-05-20] [foliantcontrib.templateparser](#) 1.0.1

- add ext\_context param for external file with context
- allow separate templates to fail, the preprocessor would issue warning and skip them

## [2019-05-20] [foliantcontrib.utils.combined\\_options](#) 1.0.4

- Add path\_convertor which converts string options to pathlib.PosixPath

## [2019-05-17] [foliantcontrib.blockdiag](#) 1.0.5

- Attributes of pseudo-XML tags have higher priority than config file options.

## [2019-05-17] [foliantcontrib.plantuml](#) 1.0.6

- Attributes of <plantuml> tag have higher priority than config file options.

## [2019-05-17] [foliantcontrib.utils.preprocessor\\_ext](#) 1.0.0

- Initial release.

## [2019-05-15] [foliantcontrib.utils.combined\\_options](#) 1.0.3

- add **iter** method to allow `for param in options`

## [2019-05-15] [foliantcontrib.utils.combined\\_options](#) 1.0.2

- add '0"1' to bool\_convertor

## [2019-05-14] [foliantcontrib.templateparser](#) 1.0.0

- Initial release

## [2019-04-30] [foliantcontrib.bindsympli](#) 1.0.12

- Capture the output of the Puppeter-based script and write it to STDOUT.

## [2019-04-15] [foliantcontrib.swaggerdoc](#) 1.1.3

- Fix issues with json and yaml. All spec files are now loaded with yaml loader.
- Change PyYAML to ruamel.yaml
- jinja mode is deprecated, widdershins is the default mode

## [2019-04-10] [foliantcontrib.mkdocs](#) 1.0.8

- Escape control characters (double quotation marks, dollar signs, backticks) that may be used in system shell commands.

## [2019-04-10] [foliantcontrib.pandoc](#) 1.0.10

- Add backticks to the set of characters that should be escaped.

## [2019-04-10] [foliantcontrib.pandoc](#) 1.0.9

- Escape double quotation marks ( ") and dollar signs ( \$) which may be used in PDF, docx, and TeX generation commands as parts of filenames, variable values, etc. Enclose filenames that may be used in commands into double quotes.

## [2019-04-05] [foliantcontrib.includes](#) 1.0.11

- Take into account the results of work of preprocessors that may be applied before includes within a single Foliant project. Rewrite the currently processed Markdown

file path with the path of corresponding file that is located inside the project source directory only if the currently processed Markdown file is located inside the temporary working directory and the included file is located outside the temporary working directory. Keep all paths unchanged in all other cases.

## [2019-03-27] [foliantcontrib.graphviz](#) 1.0.6

- Added `as_image` option.

## [2019-03-21] [foliantcontrib.anchors](#) 1.0.2

- Added `preprocessor_ext` for better warnings (and better code)

## [2019-03-21] [foliantcontrib.anchors](#) 1.0.1

- Added ‘element’ option to customize anchor span element.

## [2019-03-21] [foliantcontrib.anchors](#) 1.0.0

- Initial release

## [2019-03-21] [foliantcontrib.utils.combined\\_options](#) 1.0.1

- Add boolean convertor

## [2019-03-14] [foliantcontrib.notifier](#) 1.0.0

Initial release.

## [2019-02-21] [foliantcontrib.testrail](#) 1.1.8

- Hardcoded section headers processing removed.

## [2019-02-18] [foliantcontrib.replace](#) 1.0.2

- Now it’s possible to pass the lambda function from dictionary file.
- `with_confirmation` parameter added.

## [2019-02-15] [foliantcontrib.csvtables](#) 1.0.1

- setup.py fixed.

## [2019-02-14] [foliantcontrib.graphviz](#) 1.0.4

- Moved combined\_options out

## [2019-02-14] [foliantcontrib.apilinks](#) 1.1.3

- Moved combined\_options into a submodule

## [2019-02-14] [foliantcontrib.pgsqldoc](#) 1.1.5

- Move combined\_options into another module

## [2019-02-14] [foliantcontrib.utils.combined\\_options](#) 1.0.0

- Initial release.

## [2019-02-12] [foliantcontrib.testrail](#) 1.1.7

- Sections exclusion fixed.

## [2019-02-08] [foliantcontrib.testrail](#) 1.1.6

- Case structure output fixed if any problem occurs.

## [2019-02-01] [foliantcontrib.testrail](#) 1.1.5

- Bug with test case table numbering when deleting empty objects fixed.
- Readme updated.

## [2019-01-21] [foliantcontrib.apilinks](#) 1.1.1

- Added filename to warnings.

## [2019-01-10] [foliantcontrib.bindsympli](#) 1.0.11

- Disable images downloading from design pages only, but not from login page.

## [2018-12-24] [foliantcontrib.graphviz](#) 1.0.2

- Fixed external diagrams not reloading on change.
- Fixed external diagrams are not crashing preprocessor if the file is missing.

## [2018-12-20] [foliantcontrib.bindsympli](#) 1.0.10

- Check if the design page exists and the image URL is valid.

## [2018-12-17] [foliantcontrib.graphviz](#) 1.0.1

- Added ‘src’ tag option to load diagram source from external file.

## [2018-12-17] [foliantcontrib.graphviz](#) 1.0.0

- Initial release

## [2018-12-13] [foliantcontrib.apilinks](#) 1.1.0

- Prefixes are now case insensitive.
- Only prefixes which are defined are trimmed.
- New option `only-defined-prefixes` to ignore all prefixes which are not listed in config.
- Options renamed and regrouped. Breaks backward compatibility.
- Support of several reference pattern and properties (to catch models).
- Now search on API page for headers h1, h2, h3 and h4.

## [2018-12-06] [foliantcontrib.subset](#) 1.0.8

- Remove forgotten unnecessary import.

## [2018-12-06] [foliantcontrib.subset](#) 1.0.7

- Move the imports of the `oyaml` module directly into the methods that use it.

## [2018-12-06] [foliantcontrib.bindsympli](#) 1.0.9

- Move the `while` loop from JavaScript code to Python code.
- Add the `max_attempts` config option.
- Require Foliant 1.0.8 because of using the `utils.output()` method.

## [2018-12-04] [foliantcontrib.subset](#) 1.0.6

- Fix a bug: check if subset partial config contains `chapters` section correctly.
- Inherit the class `Cli` from `BaseCli`, not from `Cliar`.

## [2018-12-04] [foliantcontrib.multiproject](#) 1.0.9

- Inherit the class `Cli` from `BaseCli`, not from `Cliar`.

## [2018-12-04] [foliantcontrib.apilinks](#) 1.0.5

- Now both command and endpoint prefix are ensured to start from root (/).

## [2018-12-03] [foliantcontrib.apilinks](#) 1.0.4

- Fix not catching errors from `urllib`.
- Added ‘ignoring-prefix’ option.
- Added ‘endpoint-prefix’ option into API->Name section.

## [2018-11-29] [foliantcontrib.apilinks](#) 1.0.3

- Add require-prefix option.

## [2018-11-29] [foliantcontrib.apilinks](#) 1.0.2

- Trim prefixes function.

## [2018-11-29] [foliantcontrib.apilinks](#) 1.0.1

- Update docs, fix anchor error.
- Add all HTTP verbs to regular expression.

## [2018-11-27] [foliantcontrib.apilinks](#) 1.0.0

- Initial release.

## [2018-11-23] [foliantcontrib.templates.preprocessor](#) 1.0.3

- Fix `packages` value in `setup.py` of the template: use `foliant preprocessors` instead of `foliantcontrib.preprocessors`.
- Require Foliant 1.0.8 in `setup.py` of the template.

## [2018-11-20] [foliantcontrib.testrail](#) 1.1.4

- Another bug with multi-select parameter processing fixed.

## [2018-11-20] [foliantcontrib.testrail](#) 1.1.3

- Jinja templates updated.
- Bug with multi-select parameter processing fixed.

## [2018-11-19] [foliantcontrib.testrail](#) 1.1.2

- Now it's possible to use dropdown type parameters for test cases samplings.

## [2018-11-19] [foliantcontrib.testrail](#) 1.1.1

- Readme updated.

## [2018-11-19] [foliantcontrib.testrail](#) 1.1.0

- Removed parameters:
  - `platforms`,
  - `platform_id`,
  - `add_cases_without_platform`,
  - `add_unpublished_cases`.
- Added parameters:
  - `exclude_suite_ids` – to exclude suites from final document by ID,
  - `exclude_section_ids` – to exclude sections from final document by ID,

- exclude\_case\_ids – to exclude cases from final document by ID,
- add\_case\_id\_to\_std\_table - to add column with case ID to the testing table,
- multi\_param\_name - name of custom TestRail multi-select parameter for cases sampling,
- multi\_param\_select - values of multi-select parameter for cases sampling,
- multi\_param\_select\_type – sampling method,
- add\_cases\_without\_multi\_param - to add cases without any value of multi-select parameter,
- add\_multi\_param\_to\_case\_header – to add values of multi-select parameter to the case headers,
- add\_multi\_param\_to\_std\_table – to add column with values of multi-select parameter to the testing table,
- checkbox\_param\_name - name of custom TestRail checkbox parameter for cases sampling,
- checkbox\_param\_select\_type – state of custom TestRail checkbox parameter for cases sampling,
- choose\_priorities – selection of case priorities for cases sampling,
- add\_priority\_to\_case\_header - to add priority to the case header,
- add\_priority\_to\_std\_table – to add column with priority to the testing table.
- Renamed parameters:
  - add\_case\_id\_to\_case\_name -> add\_case\_id\_to\_case\_header.
- Fixed config parsing.

## [2018-11-19] [foliantcontrib.pgsqldoc](#) 1.1.3

- Add tests; refactor code
- Fix triggers and functions; add description to functions
- Fix template

## [2018-11-16] [foliantcontrib.templates.preprocessor](#) 1.0.2

- Require foliantcontrib.init 1.0.7, import the `output()` method.
- Do not rewrite source Markdown file if an error occurs.

## [2018-11-16] [foliantcontrib.multiproject](#) 1.0.8

- Do not rewrite source Markdown file if an error occurs in RepoLink preprocessor.

## [2018-11-16] [foliantcontrib.macros](#) 1.0.4

- Do not rewrite source Markdown file if an error occurs.

## [2018-11-16] [foliantcontrib.includes](#) 1.0.10

- Do not rewrite source Markdown file if an error occurs.

## [2018-11-16] [foliantcontrib.imgcaptions](#) 1.0.1

- Do not rewrite source Markdown file if an error occurs.

## [2018-11-16] [foliantcontrib.imagemagick](#) 1.0.1

- Do not rewrite source Markdown file if an error occurs.

## [2018-11-16] [foliantcontrib.flags](#) 1.0.2

- Do not rewrite source Markdown file if an error occurs.

## [2018-11-16] [foliantcontrib.epsconvert](#) 1.0.5

- Do not rewrite source Markdown file if an error occurs.

## [2018-11-16] [foliantcontrib.customids](#) 1.0.5

- Do not rewrite source Markdown file if an error occurs.

## [2018-11-16] [foliantcontrib.bindsympli](#) 1.0.8

- Do not rewrite source Markdown file if an error occurs.

## [2018-11-16] [foliantcontrib.gupload](#) 1.1.5

- Provide compatibility with Foliant 1.0.8.

## [2018-11-16] [foliantcontrib.slate](#) 1.0.4

- Provide compatibility with Foliant 1.0.8.

- Fix preprocessor: if error source won't be cleared.

## [2018-11-14] [foliantcontrib.plantuml](#) 1.0.5

- Do not rewrite source Markdown file if an error occurs.
- Use `output()` method and Foliant 1.0.8.

## [2018-11-14] [foliantcontrib.blockdiag](#) 1.0.4

- Do not rewrite source Markdown file if an error occurs.
- Use `output()` method and Foliant 1.0.8.

## [2018-11-14] [foliantcontrib.mkdocs](#) 1.0.7

- Provide compatibility with Foliant 1.0.8.

## [2018-11-14] [foliantcontrib.pandoc](#) 1.0.8

- Provide compatibility with Foliant 1.0.8.

## [2018-11-14] [foliantcontrib.init](#) 1.0.7

- Provide compatibility with Foliant 1.0.8.

## [2018-11-14] [foliant](#) 1.0.8

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

## [2018-11-14] [foliantcontrib.pandoc](#) 1.0.7

- Provide compatibility with Foliant 1.0.7.

## [2018-11-14] [foliantcontrib.mkdocs](#) 1.0.6

- Provide compatibility with Foliant 1.0.7.

## [2018-11-14] [foliant](#) 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.

## [2018-11-13] [foliantcontrib.init](#) 1.0.6

- Provide compatibility with Foliant 1.0.7.

## [2018-11-12] [foliantcontrib.multilinetables](#) 1.2.2

- Problem with deletion of table strings containing only spaces fixed (critical for lists in grid tables).

## [2018-11-09] [foliantcontrib.subset](#) 1.0.5

- Do not use `yaml` alias for `oyaml` module to prevent possible influence of this overriding on other parts of code.

## [2018-11-09] [foliantcontrib.plantuml](#) 1.0.4

- Additionally check if diagram image is not saved.

## [2018-11-09] [foliantcontrib.blockdiag](#) 1.0.3

- Do not fail the preprocessor if some diagrams contain errors. Write error messages into the log.

## [2018-11-08] [foliantcontrib.slate](#) 1.0.3

- Add slate preprocessor which copies the images outside `src` into the slate project.

## [2018-11-08] [foliantcontrib.testrail](#) 1.0.7

- Minor fixes.

## [2018-11-08] [foliantcontrib.plantuml](#) 1.0.3

- Add `parse_raw` option.
- Do not fail the preprocessor if some diagrams contain errors. Write error messages into the log.

## [2018-11-08] [foliantcontrib.testrail](#) 1.0.6

- Added: parameters to exclude suite and section headers from the final document.

## [2018-11-07] [foliantcontrib.testrail](#) 1.0.5

- Minor fixes.

## [2018-11-07] [foliantcontrib.testrail](#) 1.0.4

- Fixed: if there is only one suite in project, it's header not added to the contents.

## [2018-11-02] [foliantcontrib.gupload](#) 1.1.4

- Code refactored.

## [2018-11-01] [foliantcontrib.templates.preprocessor](#) 1.0.1

- Add `package_data` to `setup.py`.

## [2018-11-01] [foliantcontrib.gupload](#) 1.1.3

- Logger bug fixed.

## [2018-10-31] [foliantcontrib.swaggerdoc](#) 1.1.2

- Bug fixes
- All path parameters in config now accept either strings or `!path` strings

## [2018-10-31] [foliantcontrib.swaggerdoc](#) 1.1.1

- Add ‘`additional_json_path`’ param for jinja mode

- Add support for several json\_urls

## [2018-10-30] [foliantcontrib.multilinetables](#) 1.2.1

- Possibility to rewrite source files added.

## [2018-10-30] [foliantcontrib.testrail](#) 1.0.3

- Possibility to rewrite source file added.

## [2018-10-29] [foliantcontrib.bindsympli](#) 1.0.7

- Use 60-seconds timeout instead of 30-seconds. Provide multiple attempts to open pages.

## [2018-10-29] [foliantcontrib.testrail](#) 1.0.2

- Suites collecting fixed.

## [2018-10-29] [foliantcontrib.multilinetables](#) 1.2.0

- Conversion th the grid format added for arbitrary cell' content (multiple paragraphs, code blocks, lists, etc.).

## [2018-10-24] [foliantcontrib.multiproject](#) 1.0.7

- Allow to override the `edit_uri` config option of RepoLink preprocessor with the `FOLIANT_REPOLINK_EDIT_URI` system environment variable.

## [2018-10-23] [foliantcontrib.multiproject](#) 1.0.6

- Tidy up CLI arguments.

## [2018-10-23] [foliantcontrib.subset](#) 1.0.4

- Tidy up command line arguments one more time.

## [2018-10-23] [foliantcontrib.subset](#) 1.0.3

- Tidy up command line arguments.

## [2018-10-23] [foliantcontrib.subset](#) 1.0.2

- Fix a bug with object names.

## [2018-10-22] [foliantcontrib.subset](#) 1.0.1

- Parse YAML fairly. Merge config files recursively.

## [2018-10-19] [foliantcontrib.swaggerdoc](#) 1.1.0

- Change parameter names and behavior incompatible with 1.0.0
- Add conversion to md with widdershins

## [2018-10-11] [foliantcontrib.includes](#) 1.0.9

- Don't crash on failed repo sync (i.e. when you're offline).

## [2018-10-11] [foliantcontrib.mkdocs](#) 1.0.5

- Require MkDocs 1.0.4.

## [2018-10-02] [foliantcontrib.replace](#) 1.0.1

- Strings with image links are ignored.

## [2018-10-01] [foliantcontrib.gupload](#) 1.1.2

- Convert to google docs format setting added.

## [2018-09-25] [foliantcontrib.gupload](#) 1.1.1

- Unification of repository name, settings section name, and command.

## [2018-09-25] [foliantcontrib.gupload](#) 1.1.0

- Backend was converted to CLI extension.

## [2018-09-25] [foliantcontrib.multilinetables](#) 1.1.3

- ‘targets’ option added to the preprocessor settings.

## [2018-09-21] [foliantcontrib.slate](#) 1.0.2

- Rename shards\_path param to shards. It now accepts string or list.
- Fix no header param.

## [2018-09-20] [foliantcontrib.slate](#) 1.0.1

- Remove flatten. First chapter goes to index.html.md; all the rest go into the includes.

## [2018-09-18] [foliantcontrib.gupload](#) 1.0.1

- Command line authentication was added, for example for Docker use.

## [2018-09-14] [foliantcontrib.testrail](#) 1.0.1

- Preprocessor folder structure fixed.

## [2018-09-12] [foliantcontrib.bindsympli](#) 1.0.6

- Do not disable images downloading. Use delays when filling email and password fields. Wait for idle network connections when loading pages.

## [2018-08-31] [foliant](#) 1.0.6

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

## [2018-08-29] [foliantcontrib.pgsqldoc](#) 1.1.2

- Queries are now ordered (not adjustable right now)

- Flexible filters instead of strict filtering by schema

## [2018-08-27] [foliantcontrib.pgsqldoc](#) 1.1.1

- Fix scheme template (blank lines issue)
- Refactor queries code

## [2018-08-24] [foliantcontrib.multilinetables](#) 1.1.2

- Now it's possible to break the text anywhere in multiline tables with custom tag.
- Fixed determination of columns number in tables with and without side lines.

## [2018-08-24] [foliantcontrib.pgsqldoc](#) 1.1.0

- Docs and scheme structure is now defined by Jinja2 templates.

## [2018-08-22] [foliantcontrib.multilinetables](#) 1.1.1

- Bug with regular expression fixed. 3+ code strings with || operator in a row are not perceived as a tables now.

## [2018-08-22] [foliantcontrib.multilinetables](#) 1.1.0

- Code strings with || operator are not perceived as a tables now.

## [2018-07-31] [foliantcontrib.bump](#) 1.0.2

- Declare semver as dependency.

## [2018-07-29] [foliantcontrib.bump](#) 1.0.1

- Fix packaging with setup.py. Poetry doesn't quite do the trick 😞

## [2018-07-28] [foliantcontrib.bump](#) 1.0.0

Initial release.

## [2018-07-24] [foliantcontrib.mkdocs](#) 1.0.4

- Provide customizable default names for untitled nested groups of chapters.

## [2018-07-24] [foliantcontrib.flatten](#) 1.0.4

- Skip empty headings of nested subsections.

## [2018-07-23] [foliantcontrib.includes](#) 1.0.8

- Require at least one space after hashes in the beginning of each heading.
- Add `inline` option to the `<include>` tag.
- Fix the bug: do not ignore empty lines after headings when using `sethead`.
- Fix the bug: allow to use less than 3 characters in the heading content.
- Do not mark as headings the strings that contain more than 6 leading hashes. If shifted heading level is more than 6, mark the heading content as bold paragraph text, not as heading.

## [2018-06-08] [foliantcontrib.multiproject](#) 1.0.5

- Provide Git submodules support.

## [2018-06-07] [foliantcontrib.flatten](#) 1.0.3

- Use flattened file path in `includes` preprocessor call.
- Require `includes` preprocessor 1.0.7.

## [2018-06-06] [foliantcontrib.includes](#) 1.0.7

- Fix paths resolving in case of recursive processing of include statements.
- Allow revision markers in repo aliases.

## [2018-06-04] [foliantcontrib.includes](#) 1.0.6

- Fix logging in file search method.
- Fix top heading level calculation.

## [2018-06-04] [foliantcontrib.multiproject](#) 1.0.4

- Provide compatibility with Foliant 1.0.5. Allow to use multiple config files.

## [2018-06-04] [foliantcontrib.pandoc](#) 1.0.6

- Apply `flatten` after all preprocessors, not before them. This fixes incompatibility with `foliantcontrib.includes` 1.0.5.

## [2018-06-04] [foliantcontrib.flatten](#) 1.0.2

- Fix incorrect `includes` preprocessor call.
- Require Foliant 1.0.5.

## [2018-06-04] [foliantcontrib.init](#) 1.0.5

- Require Foliant 1.0.5 with `prompt_toolkit^2.0.0`.

## [2018-05-30] [foliantcontrib.customids](#) 1.0.4

- Provide separate block-level HTML elements for the anchors. Allow to define custom stylesheets for these elements.

## [2018-05-25] [foliantcontrib.includes](#) 1.0.5

- Use paths that are relative to the current processed Markdown file.
- Fix `sethead` behavior for headings that contains hashes ( #).

## [2018-05-14] [foliant](#) 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.

## [2018-05-10] [foliantcontrib.pandoc](#) 1.0.5

- Add `slug` config option.

## [2018-05-08] [foliantcontrib.multiproject](#) 1.0.3

- Fix config loading. Other small fixes.

## [2018-04-25] [foliantcontrib.multiproject](#) 1.0.2

- Fix bugs with the project directory path and Git repos syncronizing.

## [2018-04-23] [foliantcontrib.multiproject](#) 1.0.1

- Fix logging.

## [2018-04-20] [foliantcontrib.bindsympli](#) 1.0.5

- Add logging.

## [2018-04-20] [foliantcontrib.plantuml](#) 1.0.2

- Fix logging in `__init__`.

## [2018-04-20] [foliantcontrib.plantuml](#) 1.0.1

- Add logging.

## [2018-04-20] [foliantcontrib.flatten](#) 1.0.1

- Fix incorrect `includes` preprocessor call.
- Add logging.
- Require Foliant 1.0.4.

## [2018-04-19] [foliantcontrib.epsconvert](#) 1.0.4

- Do not use image path when computing MD5 hash.
- Add `targets` config option.
- Add logging.

## [2018-04-19] `foliantcontrib.templates.preprocessor` 1.0.0

- Initial release.

## [2018-04-18] `foliantcontrib.customids` 1.0.3

- Add `targets` config option.
- Add logging.

## [2018-04-14] `foliantcontrib.blockdiag` 1.0.2

- Add logging.
- Require Foliant 1.0.4.

## [2018-04-14] `foliantcontrib.pandoc` 1.0.4

- Add logs.
- Update for Foliant 1.0.4: Pass logger to spinner.
- Require Foliant 1.0.4.

## [2018-04-14] `foliantcontrib.mkdocs` 1.0.3

- Add logs.
- Update for Foliant 1.0.4: Pass logger to spinner.
- Require Foliant 1.0.4.

## [2018-04-14] `foliantcontrib.init` 1.0.4

- Replace placeholders in file and directory names.
- Process `*.py` files.
- User Template strings instead of format strings for safer substitutions.
- Update for Foliant 1.0.4: Pass logger to spinner.
- Require Foliant 1.0.4.

## [2018-04-11] [foliant](#) 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 it easier to call it from extensions.

## [2018-04-10] [foliantcontrib.bindsympli](#) 1.0.4

- Describe the preprocessor usage in `README.md`.

## [2018-04-10] [foliantcontrib.bindsympli](#) 1.0.3

- Eliminate external Perl scripts, rewrite the preprocessor code in Python.

## [2018-04-02] [foliant](#) 1.0.3

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

## [2018-04-01] [foliantcontrib.includes](#) 1.0.4

- Fix the pattern for headings detection.

## [2018-03-31] [foliantcontrib.includes](#) 1.0.3

- Allow hashes (# characters) in the content of headings.

## [2018-03-29] [foliantcontrib.epsconvert](#) 1.0.3

- Take into account the content of image file when computing MD5 hash.

## [2018-03-29] [foliantcontrib.epsconvert](#) 1.0.2

- Add support of any local paths. Add image cache.

- Remove `mogrify_path` and `diagrams_cache_dir` options, add `convert_path` and `cache_dir` instead.

## [2018-03-28] [foliantcontrib.customids](#) 1.0.2

- Process first heading and all other headings separately.

## [2018-03-27] [foliantcontrib.customids](#) 1.0.1

- Update README.md and docstrings.
- Update long description content type in setup.py

## [2018-03-27] [foliantcontrib.bindsympli](#) 1.0.2

- Change the path for non-Python scripts once more.

## [2018-03-27] [foliantcontrib.bindsympli](#) 1.0.1

- Change the path for non-Python scripts.

## [2018-03-21] [foliantcontrib.includes](#) 1.0.2

- Fix inappropriate translation of image URLs into local paths.

## [2018-03-21] [foliantcontrib.mkdocs](#) 1.0.2

- Add `use_headings` and `slug` options for MkDocs backend.
- Fix inappropriate translation of image URLs into local paths in MkDocs preprocessor.

## [2018-03-17] [foliant](#) 1.0.2

- Use README.md as package description.

## [2018-03-13] [foliantcontrib.epsconvert](#) 1.0.1

- Add `diagrams_cache_dir` option support.

## [2018-02-28] [foliantcontrib.pandoc](#) 1.0.3

- Change Pandoc command line parameter `--reference-docx` to `--reference-doc`.

## [2018-02-25] [foliant](#) 1.0.1

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

## [2018-02-23] [foliant](#) 1.0.0

- Complete rewrite.

## [2018-02-16] [foliantcontrib.blockdiag](#) 1.0.1

- Add `pdf` output format support.

## [2018-02-07] [foliantcontrib.init](#) 1.0.3

- Upon creation, relative path to the created project directory is returned instead of an absolute one.
- Templates: basic: Foliant docs related content removed from `README.md`.
- Templates: basic: `foliantcontrib.mkdocs` added to `requirements.txt`.

## [2018-02-07] [foliantcontrib.init](#) 1.0.2

- Add `slug` placeholder.
- Process placeholders in `.yml`, `.txt`, and `.md` files, not just `foliant.yml`.
- Templates: basic: Add `Dockerfile`, `docker-compose.yml`, `requirements.txt`, and `README.md`.

## [2018-02-07] [foliantcontrib.init](#) 1.0.1

- Fix issue with `init` command missing after installation.
- Fix issue with missing templates after installation.

## [2018-02-01] [foliantcontrib.macros](#) 1.0.3

- Add tag `<m>...</m>`.

## [2018-01-17] [foliantcontrib.macros](#) 1.0.2

- Switch from unnamed to named parameters.
- Macro name is now defined in the tag body instead of “name” option.

## [2018-01-15] [foliantcontrib.macros](#) 1.0.1

- Preserve param case.

## [2018-01-06] [foliantcontrib.flags](#) 1.0.1

- Add `targets` and `backends` options to `<if>` tag.

## [2018-01-05] [foliantcontrib.pandoc](#) 1.0.2

- Change default Markdown flavor from `markdown_strict` to `markdown`.

## [2017-12-17] [foliantcontrib.pandoc](#) 1.0.1

- Add `tex` target.

## [2017-12-16] [foliantcontrib.mkdocs](#) 1.0.1

- Add `ghp` target for GitHub Pages deploy with `mkdocs gh-deploy`.

## [2017-12-15] [foliantcontrib.includes](#) 1.0.1

- Fix git repo name detection when the repo part contains full stops.