

Foliant

User's Manual

Welcome to Foliant!

Foliant is an all-in-one single-source documentation authoring tool. It lets you produce standalone documents in **pdf** and **docx**, build **static websites** and upload pages to **Confluence**, all from single Markdown source.

Foliant is a higher order tool, which means it uses other programs to do its job. For building pdf and docx, it can use [Pandoc](#) or [md-to-pdf](#), for websites [MkDocs](#), [Aglio](#) or [Slate](#).

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

Foliant is highly extensible, so if it lacks some functions or output formats you can always make a plugin for it or request one from our team.

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

Who Is It for?

You'll love Foliant if you:

- need to ship documentation as pdf, docx, and websites
- 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 would rather use Markdown
- want a tool that you can extend with custom plugins without dealing with something as over-engineered as Sphinx
- want to work with docs as code and make them a part of your CI pipeline
- have a lot of segregated repositories with sources of your documents and want to standardize the documentation approach.

Changelog

Here is the changelog of [Foliant Core](#), the main and only strictly required package. See also the history of releases of numerous Foliant extensions.

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.

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

Installation of Foliant is 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:

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

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

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

Finally, install the Pandoc backend:

```
$ pip3 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
```

Finally, install the Pandoc backend:

```
$ pip3 install foliantcontrib.pandoc
```

Ubuntu

1. Install Python 3 with apt.

On 18.04 or higher Python 3 will already be installed. Check that by running:

```
$ python3
```

If it is not installed, here's a way to install the latest version:

```
1 $ sudo apt update
2 $ sudo apt install software-properties-common
3 $ sudo add-apt-repository ppa:deadsnakes/ppa
4 $ sudo apt install python3.9 python3-pip
```

2. Install Foliant with pip:

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

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

```
1 $ sudo apt update
2 $ sudo apt install -y texlive-full librsvg2-bin pandoc
```

Finally, install the Pandoc backend:

```
$ pip3 install foliantcontrib.pandoc
```

Docker

There is a selection of Docker images for Foliant in the [Docker hub](#):

- `foliant/foliant:slim` – minimal image of Foliant with no extensions;
- `foliant/foliant` – the basic image with just Foliant core and the `init` command;

- `foliant/foliant:pandoc` – asic image with the addition of TexLive and Pandoc for building PDF and DOCX;
- `foliant/foliant:full` – the full image with all official Foliant extensions and third-party tools required for them to work.

Choose the image you want and run the `docker pull` command

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

If you are new to Docker, check our tutorial on using Foliant with Docker.

Quickstart

If you don't have Foliant installed, please follow the instructions first.

Step 1. Create a new project

```
$ foliant init
```

Or with Docker

```
$ docker run --rm -it --user $(id -u):$(id -g) -v $(pwd):/usr/src/app -w /usr/src/app foliant/foliant init
```

Step 2. cd into the folder created by command

```
$ cd my-project
```

Step 3. Edit the Markdown source of your documentation located in `src/index.md`.

To build a static site with [MkDocs](#), install the MkDocs backend (skip this step if you are using Docker)

```
pip3 install foliantcontrib.mkdocs
```

Step 4. Build the site with `foliant make` command

```
$ foliant make site
```

Or with Docker

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

Done! Your site is generated in the `My_Project-2020-05-25.mkdocs` folder, crank up a webserver to take a look at it

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

Now let's build a DOCX out of the same source. You will need [Pandoc](#) and Pandoc backend for that, the instructions for installing them are in the installation guide.

Step 5. Build docx

```
$ foliant make docx
```

With Docker you will need to adjust the Dockerfile first, replace the first line with the following

```
1 - FROM foliant/foliant  
2 + FROM foliant/foliant:pandoc
```

and rebuild the image

```
$ docker-compose build
```

Finally, run the make command inside the container

```
$ docker-compose run --rm foliant make docx
```

Done! The My_Project-2020-05-25.docx is created in the project dir.

If you want to know more about how Foliant works, check out the Architecture And Basic Design Concepts or just dive straight into Your First Foliant Project.

Tutorials

Your First Foliant Project

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.

It is recommended to run Foliant through Docker to get consistent results on different machines, but it's also perfectly fine to run it natively (e.g. as a pure CLI tool without virtualization). In this tutorial, we will show the example commands for both native way (these will go first) and the Docker way (these will follow).

Create New Project

All Foliant projects must adhere to a certain structure. Luckily, you don't have to memorize it thanks to the `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 --user $(id -u):$(id -g) -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
```

The `init` command created a structure for the Foliant project in `hello-foliant` subfolder.

```
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 Configuration file.

`src` is the directory for your Markdown documents. Currently, there's just one file there called `index.md`.

`requirements.txt` lists the Python packages required for the project: Foliant backends and preprocessors, MkDocs themes, and whatnot. When the Docker image for the project is built, these requirements will be 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, but in short, backends are Foliant modules responsible for converting Markdown sources into the final documentation format.

Let's start with **MkDocs** backend. First, install it using the following command

```
pip3 install foliantcontrib.mkdocs
```

Docker users would normally need to add this package to the `requirements.txt` file instead, but mkdocs is already there by default if you used `init` to generate project structure.

To build a site, 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 in your web browser. You should see something like this

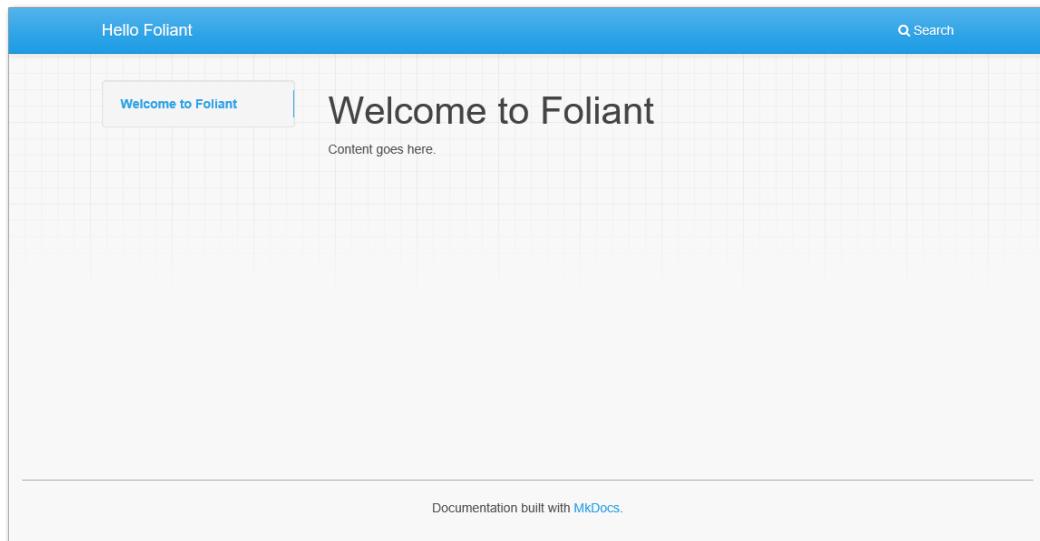


Figure 1. Basic Foliant project built with MkDocs

Build PDF

To build PDF with Pandoc natively, first you will need to install Pandoc itself and TeXLive, check Foliant installation page for instructions.

Then, 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

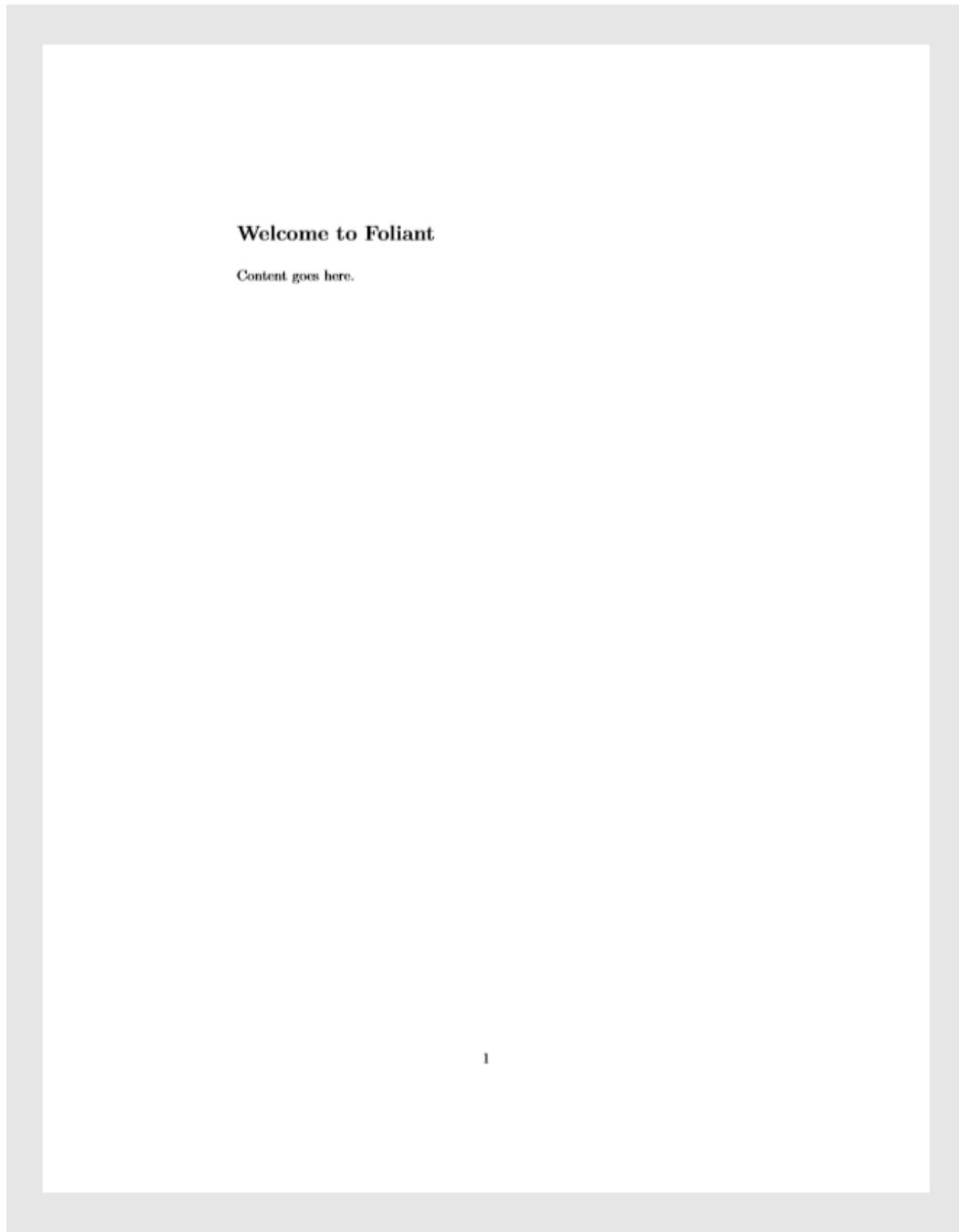


Figure 2. Basic Foliant project built with Pandoc

Edit Content

Your project's content lives in `.md` files inside the `src` folder. You can organize it into multiple files and subfolders inside the `src` as you please.

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

Let's create a file `hello.md` inside `src` folder

```
$ touch src/hello.md
```

And fill it with some content. For example

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

Now you have two files (or [chapters](#)) inside `src`, but Foliant knows only about one of them. To add `hello.md` to the project, open `foliant.yml` and add the new chapter to the `chapters` list

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

Let's rebuild the project to see the new page.

The native command

```
1 foliant make pdf && foliant make site  
2 Parsing config... Done  
3 Applying preprocessor flatten... Done  
4 Applying preprocessor _unescape... Done  
5 Making pdf with Pandoc... Done—————  
6  
7 Result:
```

```
8 Hello_Foliant-2020-05-25.pdf
9 Parsing config... Done
10 Applying preprocessor mkdocs... Done
11 Applying preprocessor _unescape... Done
12 Making site with MkDocs... Done—————
13
14 Result: Hello_Foliant-2020-05-25.mkdocs
```

The command for Docker

```
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

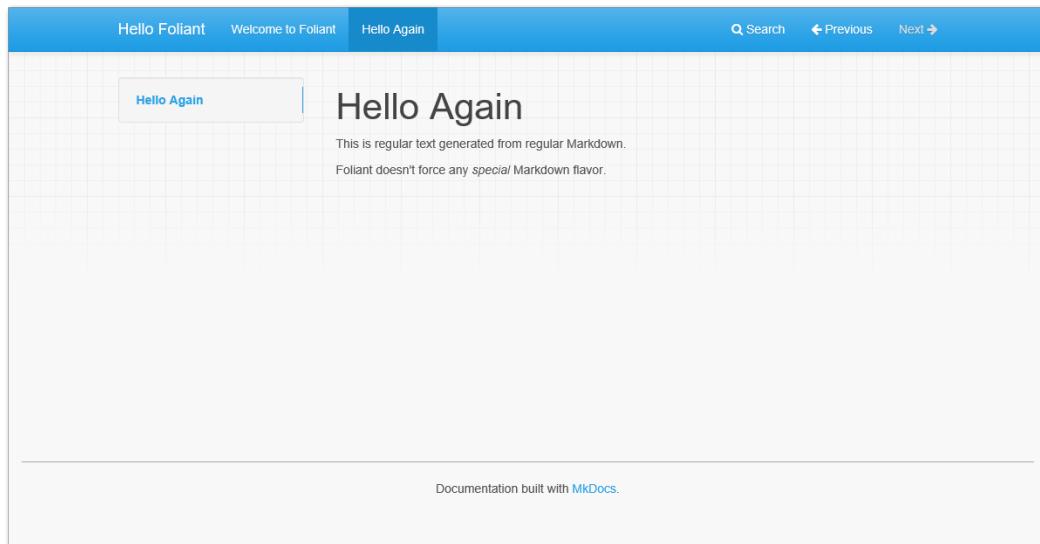


Figure 3. New page on the site

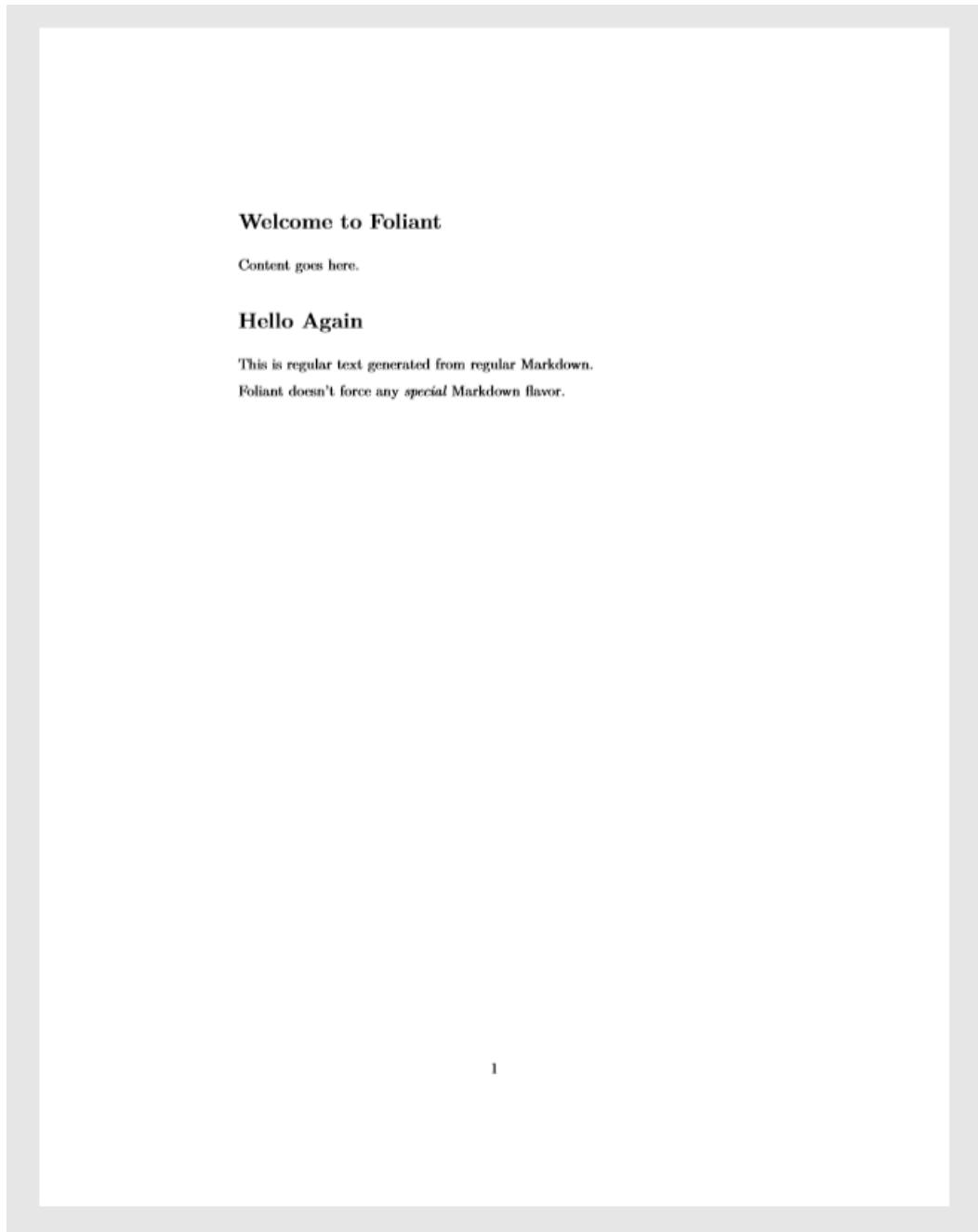


Figure 4. New page in the pdf document

Use Preprocessors

Preprocessors are additional Foliant packages that transform your Markdown chapters in different ways. You can do all kinds of stuff with them:

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

Preprocessors don't touch your sources in the `src` folder. Instead, they copy them into a temporary directory and transform the fresh copies on each build.

In fact, you have already used two preprocessors! Look at the output of the `foliant make` commands and note the lines `Applying preprocessor mkdocs` and `Applying preprocessor flatten`. The `mkdocs` preprocessor made your files compatible with MkDocs' requirements, and the `flatten` preprocessor was used to squash the project source into one file to produce a single PDF with Pandoc. These preprocessors were called by MkDocs and Pandoc backends implicitly.

Now let's add a preprocessor into the pipeline ourselves. We've chosen `Blockdiag` preprocessor for this tutorial.

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

```
$ pip3 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 project config. Open `foliant.yml` and add the following lines

```
1 title: Hello Foliant
2 +
3 + preprocessors:
4 +   - blockdiag
```

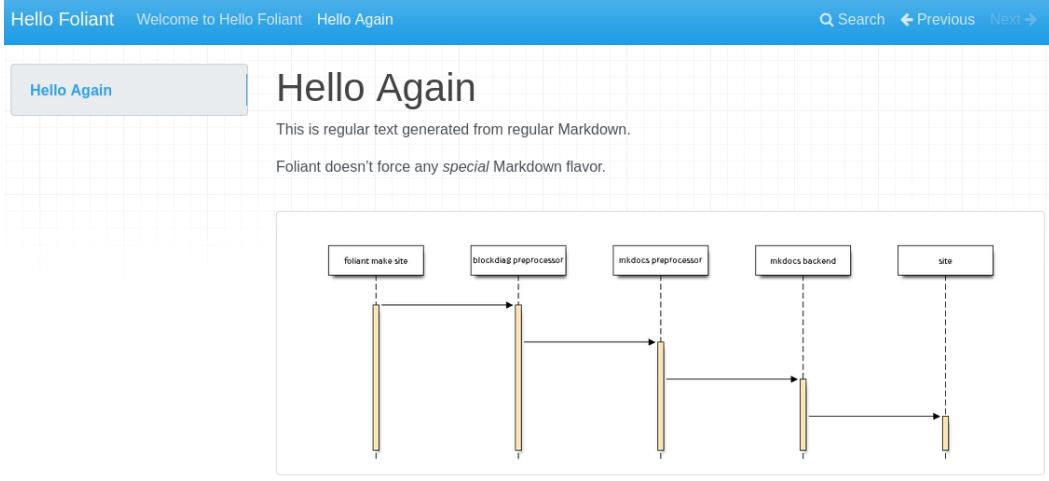
```
5  
6 chapters:  
7   - index.md  
8   - hello.md
```

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" -> "blockdiag preprocessor" -> "  
6 +       mkdocs preprocessor" -> "mkdocs backend" -> site;  
7 +   }  
7 + </seqdiag>
```

Blockdiag preprocessor extends the Markdown syntax of your documentation by adding several tags. Each tag produces a different 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 attributes.

Rebuild the site with `foliant make site` or `docker-compose run --rm foliant make site` and open it in the browser



Documentation built with [MkDocs](#).

Figure 5. Sequence diagram drawn with seqdiag on the site

Rebuild the pdf and see that the diagram is there too

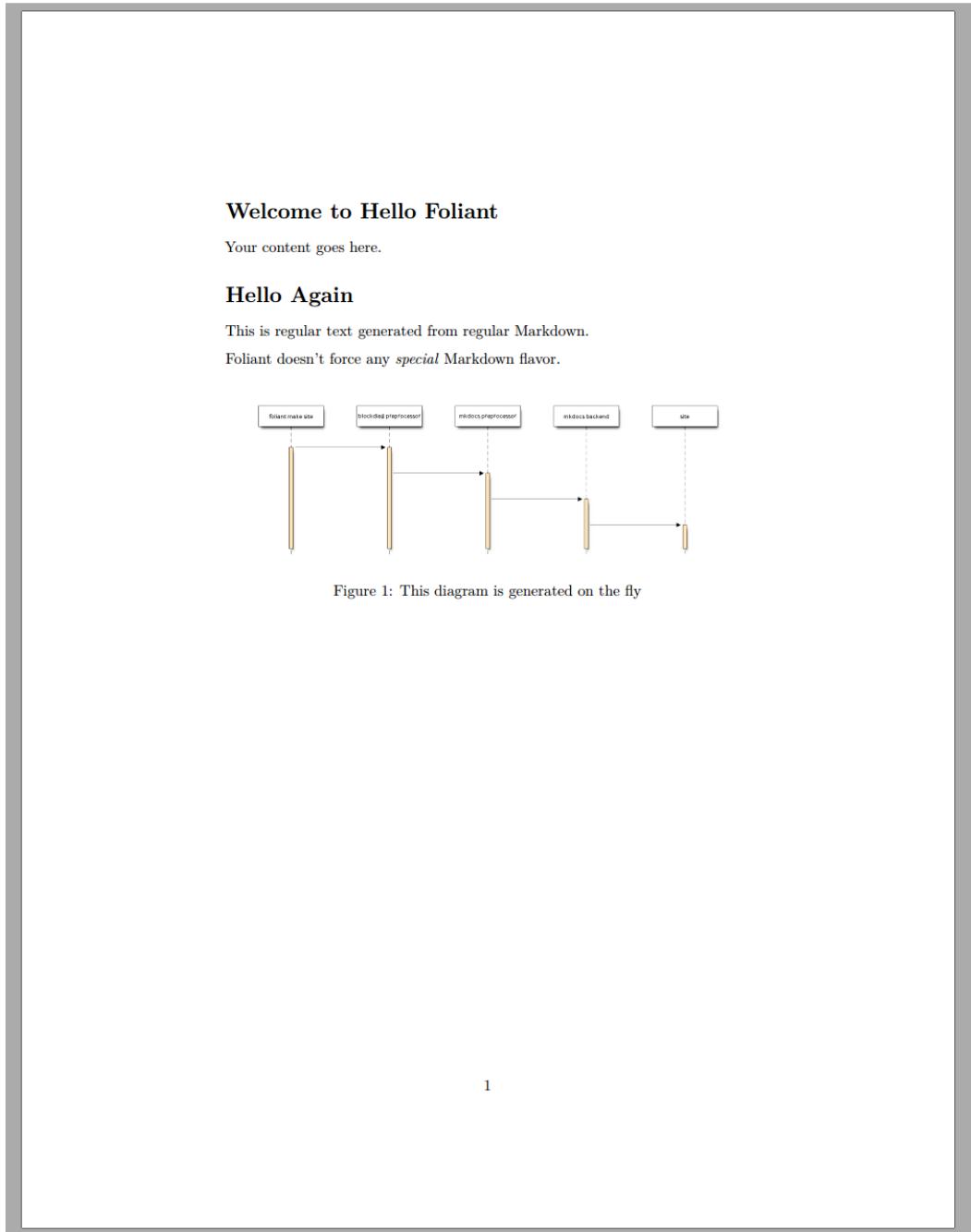


Figure 6. Sequence diagram drawn with seqdiag in the pdf

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

Hello Again

This is regular text generated from regular Markdown.

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

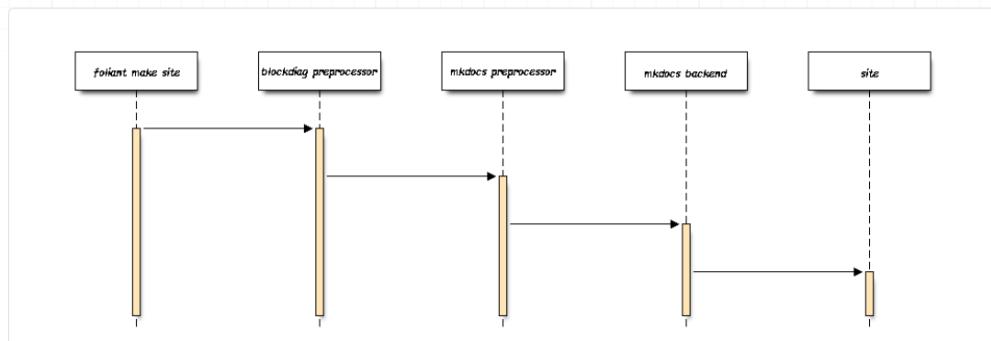


Figure 7. Sequence diagram with Comic Sans in labels, site

Hello Again

This is regular text generated from regular Markdown.

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

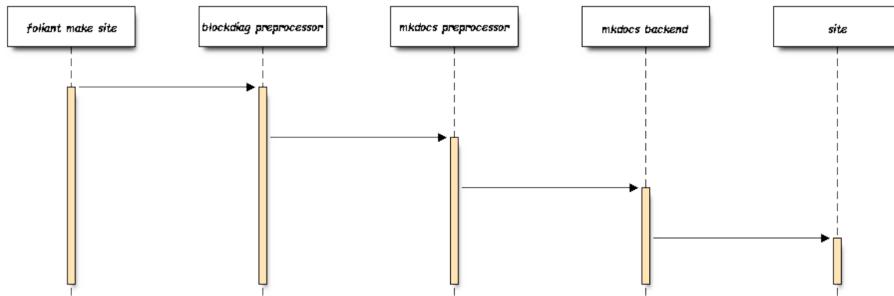


Figure 1: This diagram is generated on the fly

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

There are many more params you can define for your diagrams. You can override global params for 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="site">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, a 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.

Running Foliant in Docker

Foliant's design philosophy says that everybody should do what they do best. We don't aim to create a universal text processing combine which covers all needs of a technical writer by itself. Instead Foliant introduces integrations with different beautiful open source tools which specialize on a little chunk of work and do it perfectly.

This approach comes with a disadvantage that you have to install each one of the tools you are using in the project for Foliant to work. You may do it once on your machine but Foliant projects usually need to work as well for other people if they decide to clone your project's repository. And these people may not have the right tool installed, or they may have another version of it, or even an operating system which doesn't have the tool at all.

Docker solves this problem by creating a virtual environment which will be consistent among different machines and even different operating systems. All the required tools will be installed and configured inside this virtual environment so all it's left to do is to run the build.

Working with Docker may seem complicated for non-programmers, but we will try to make it simple. We will concentrate on practical examples and keep the technical details out of this tutorial. If you want them – check the [Docker documentation](#).

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

Follow the [instructions](#) for your Linux distribution on the official website.

After that [install Docker Compose](#).

MacOs

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

Creating a Test Project

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

If you have Foliant installed on your system, run the `init` command

```
$ foliant init
```

Type the name of the project and `cd` into the freshly created folder.

But that's the beauty of the Docker way, you don't even need to have Foliant installed on your computer to build Foliant projects. Instead of using `init` you can clone the [Foliant Project template](#). It's an empty Foliant project with the required file and directory structure, including necessary Docker configs. It's similar to what you get by running `foliant init`.

Now `cd` to the cloned directory and remove the `.git` folder, which still points to the template repository.

Setting up Docker configs

We've got a basic project which we already can build with Docker.

Inside the project dir run:

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

Right now our project can only build an MkDocs site. If we try to build a pdf we will get

```
1 $ docker-compose run --rm foliant make pdf
2 No backend available for pdf.
```

If you are already familiar with Foliant you know that to build PDFs you need to install [Pandoc](#) with TeXLive and Pandoc backend. So how do we install that in a Docker container?

In your project root you have a `Dockerfile`. This file describes the steps required to set up your virtual container.

If you open the `Dockerfile`, you will see that apart from comments it contains three lines:

```
1 FROM foliant/foliant
2
3 COPY requirements.txt .
4
5 RUN pip3 install -r requirements.txt
```

The first line means that we start out with the base Foliant image, more on that later. The second one copies the file `requirements.txt` from your project folder into the container and the last one installs all Python dependencies from this file with `pip` from inside the container.

`requirements.txt` is a file where all your Python dependencies for the project live. It means that adding the Pandoc backend to the virtual environment is a matter of adding it into `requirements.txt`. Let's do this now:

```
1 foliantcontrib.mkdocs
2 + foliantcontrib.pandoc
```

With Pandoc and TeXLive it's not that easy, because they are not Python packages. But don't worry, it's still easy enough.

Your virtual container is based on `foliant/foliant` image, which is in turn based on Ubuntu operating system. So all you need to do is to find the right commands to install the required packages as if you were on Ubuntu.

These commands are

```
1 apt update
2 apt install -y texlive-full librsvg2-bin
3 apt install -y pandoc
```

So we take this commands and put them in our `Dockerfile`, but we will put `RUN` before each one to explain Docker our intentions. The order in which lines appear in the `Dockerfile` is important. Docker does a nice job of caching stages of container build, so make a rule of putting the commands which less prone to change at the start.

In our case we will probably be editing our `requirements.txt` further down the line, but the pandoc installation commands are unlikely to change so we put them first:

```
1 FROM foliant/foliant
2
3 + ENV DEBIAN_FRONTEND=noninteractive
4 + RUN apt update
5 + RUN apt install -y texlive-full librsvg2-bin
6 + RUN apt install -y pandoc
7
8 COPY requirements.txt .
9
10 RUN pip3 install -r requirements.txt
```

We've also added an environment variable `DEBIAN_FRONTEND` which is required to install `texlive-full` inside a Docker container. Consider it magic, just don't forget to add it each time you install `texlive` in Docker.

Now we need to rebuild our container. If we were to run the `docker-compose run` command now, it would still run in the old container, which doesn't have `pandoc`. So let's build it

```
$ docker-compose build
```

This command will now take time to complete because of the TeXLive engine which is **HUGE**. Don't worry, you will need to wait for so long just once.

Now, as it's starting to get dark and you can hear the workers coming home from the factories, our image has finished building.

Let's make a PDF!

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

If all went right you will see a PDF file in your project folder.

Using different Foliant Docker images

You've noticed that `docker-compose build` command took a lot of time to complete because it needed to download and install the massive TexLive engine. It would be a pain to repeat this for each new Foliant project.

Luckily, Foliant offers [a selection of Docker images](#), each of which offers different number of tools preinstalled. One of the images is called `pandoc` and has the same packages which we've installed in the previous section.

The full list is:

- `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.

Let's update our project to use the `pandoc` image instead of manually installing the dependencies.

The image to use for the project is specified on the first line of the `Dockerfile`. Open it and replace the first line with:

```
1 - FROM foliant/foliant  
2 + FROM foliant/foliant:pandoc
```

Now remove the lines which we've added previously so your `Dockerfile` looks like this:

```
1 FROM foliant/foliant:pandoc  
2  
3 RUN pip3 install -r requirements.txt
```

Next, remove the Pandoc backend from `requirements.txt` as it is also preinstalled in the `pandoc` image.

Finally, rebuild the image and run the PDF making command:

```
1 $ docker-compose build  
2 $ docker-compose run --rm foliant make pdf
```

Once the `pandoc` image is downloaded, the build commands will always run very fast.

Working with Foliant full image

We've learned how to use Foliant with Docker, how to install dependencies inside the container and how to use different Foliant images.

Now it's time to learn about the `full` Foliant image. This is the most powerful one of all. It has all official Foliant extensions and all their dependencies preinstalled.

Once you base your `Dockerfile` on this image you will have whole power of Foliant at your disposal whenever you need it.

To use it replace the first line of your `Dockerfile` with

```
1 - FROM foliant/foliant  
2 + FROM foliant/foliant:full
```

and run the build command

```
$ docker-compose build
```

Now you can for instance make a [Slate](#) static website with your docs instead of MkDocs.

The command is

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

We had to add a `--with slate` argument to our command to specify the backend to build `site` target with. `full` Foliant docker image contains all available official backends for Foliant and several of them are capable of building the `site` target. Without the `--with` argument (or `-w` for short) Foliant would prompt you for the specific backend name interactively.

Summary

That's all you need to know to work with Foliant the Docker way. Just remember the steps:

- put your Python dependencies in the `requirements.txt` file,
- add commands for installing your non-Python dependencies into the `Dockerfile`, preceding them with the `RUN`,
- rebuild your image with `docker-compose build` command every time you edit `Dockerfile` or `requirements.txt`. No need to run it after editing your Markdown sources or Foliant-related configuration files.

And one last note for the `full` image users. We keep constantly updating Foliant, adding and updating its extensions. To use all the fresh features update the image every once in a while with command

```
$ docker pull registry.itv.restr.im:5000/foliant:full
```

And don't forget to rebuild your project's image after updating:

```
$ docker-compose build
```

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
4       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:

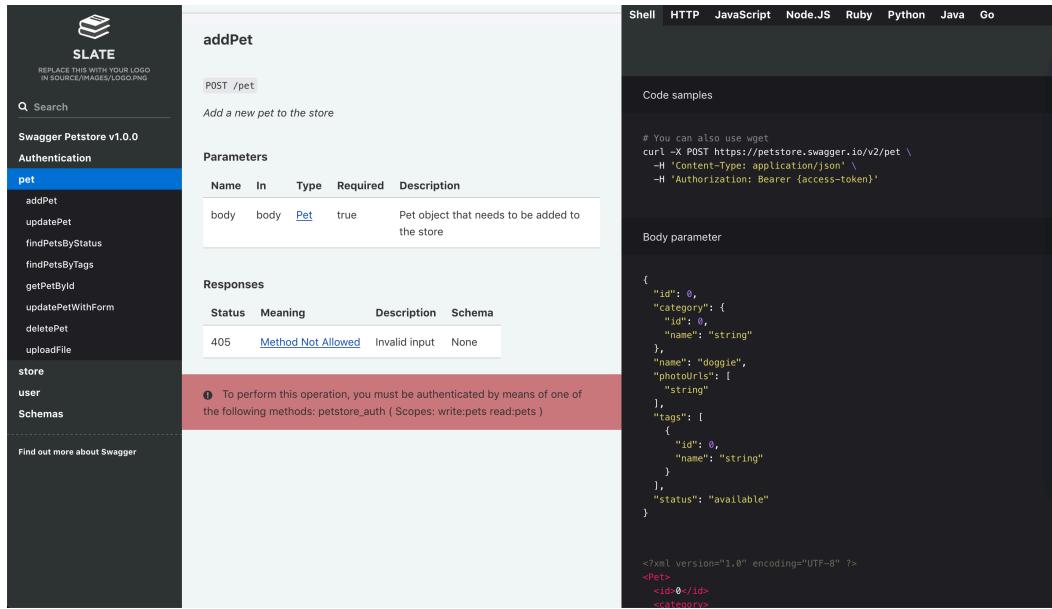


Figure 9. Slate static site

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:

The screenshot shows a static website generated by Foliant using the Slate theme. The left sidebar contains a navigation menu with items like 'SLATE' (placeholder for logo), 'Optimyze API Documentation', 'POST /read/records/(table)', 'POST /adjust', 'Types', 'SortingInfo' (selected), 'PageInfo', 'Filter', 'RecordInfo', 'ColumnValues', 'RowIdentifier', 'DatabaseRecord', 'Records', 'AddedRows', 'EditedRows', 'DeletedRows', 'Tables', 'AdjustmentInfo', and 'Error'. The main content area has two sections: 'PageInfo' and 'Filter'. The 'PageInfo' section has a 'Properties' table:

Name	Type	Required	Description
page_number	integer	true	
page_size	integer	true	

The 'TYPE DEFINITION' for PageInfo is:

```
{
  "name": "PageInfo",
  "type": "object",
  "description": "Accepts page information",
  "properties": {
    "page_number": {
      "type": "integer",
      "name": "page_number",
      "displayName": "page_number",
      "typePropertyKind": "TYPE_EXPRESSION",
      "required": true
    },
    "page_size": {
      "type": "integer",
      "name": "page_size",
      "displayName": "page_size",
      "typePropertyKind": "TYPE_EXPRESSION",
      "required": true
    }
}
```

The 'Filter' section has a 'Properties' table:

Name	Type	Required	Description
...

The 'TYPE DEFINITION' for Filter is:

```
{
  "name": "Filter",
  "type": "object",
  "description": "Filtering information",
  "properties": {
    ...
  }
}
```

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.

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:

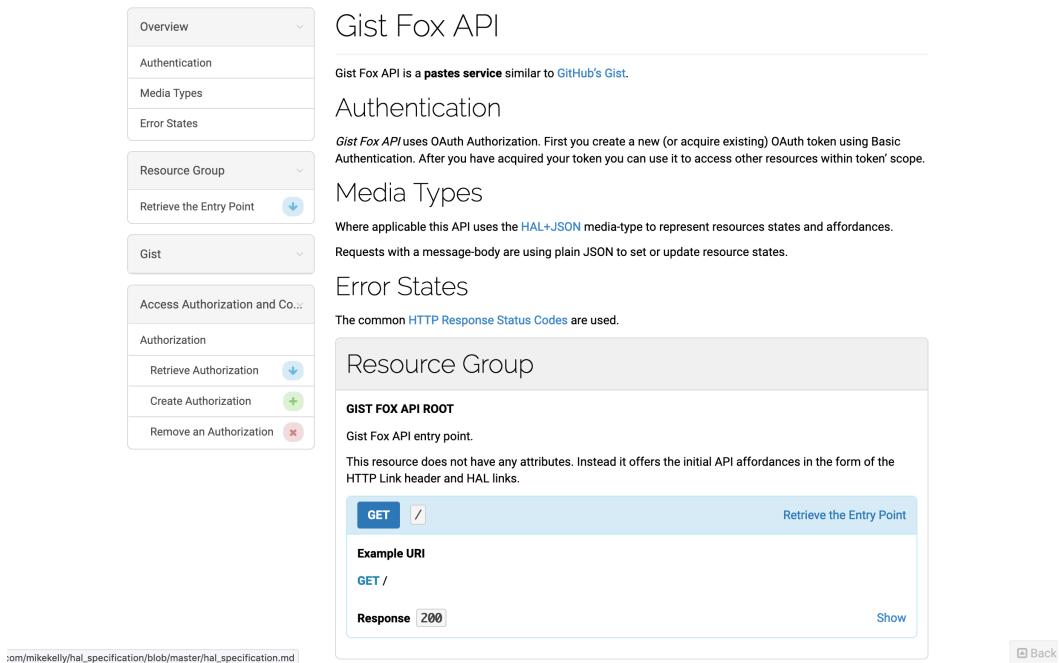


Figure 11. Aglio static site

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:

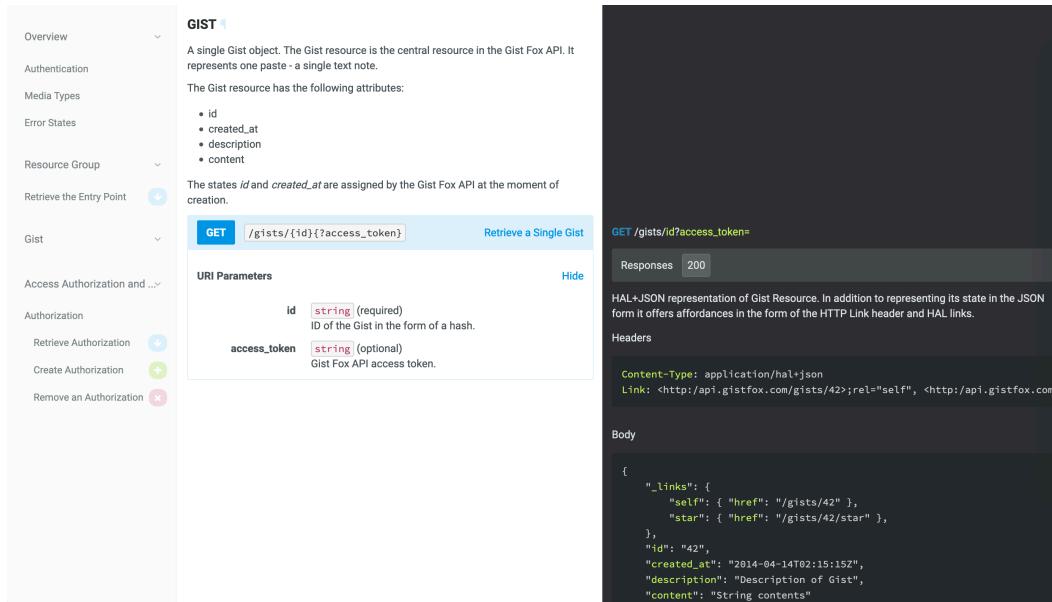


Figure 12. Aglio more beautiful static site

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**, **MySQL**, **MS SQL Server** and **DBML**. If your DBMS is not in the list, [send us an issue](#), we'll do our best to add the support to it as fast as possible.

The principles

Generally, we want to keep our docs as close to the code as possible. For documenting the source code of an application 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 -- comment) 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 information for our document. For this we will use Foliant DBDoc preprocessor. It queries the database to get its structure (including the comments) and converts it into Markdown. We can then use this Markdown to generate a static site for our documentation.

The tutorials

First tutorial is about creating a document out of [DBML](#) schema. DBML is not a database but a simplified language for describing databases.

The other two tutorials focus on the process of documenting actual databases. We have tutorials for PostgreSQL and Oracle. But the process of documenting other databases is quite similar.

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 the designing stage of your database. You can create your table structure without messing with cumbersome SQL in a more 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 it 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 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.

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 also 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 `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: !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 the 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 point Foliant the place in the Markdown source files where the generated documentation should be inserted. 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 are finished, 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 at the results:

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 out of your schema. After the 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 to the instructions in Slate backend documentation.

Documenting Oracle Database

Please note that this article covers only the basic usage of the tools.

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

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 will just need the last paragraph in this section.

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

Install Python connector for Oracle database.

```
$ pip3 install cx_Oracle
```

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.

If you are using Docker, you will 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.

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 +      dbms: oracle
9 +      host: localhost
10 +     port: 1521
11 +     dbname: orcl
12 +     user: hr
13 +     password: oracle
14 + - plantuml
15 +
```

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 `platnum.jar` in preprocessor settings like this:

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

Finally, we need to point Foliant the place in the Markdown source files where the generated documentation should be inserted. 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 +
```

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
```

```

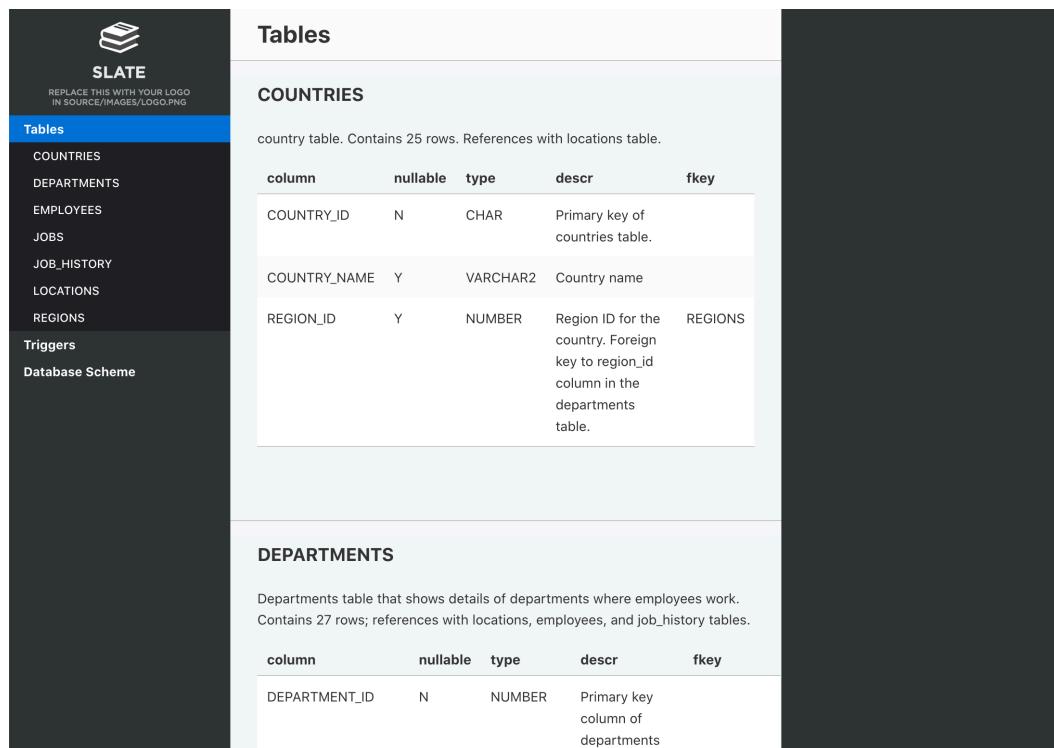
5 Applying preprocessor flatten... Done
6 Applying preprocessor _unescape... Done
7 Making site... Done
8 ...
9
10 Result: Database_Docs-2020-06-03.slate/

```

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 Foliant Slate backend interface. On the left, there's a sidebar with a logo placeholder, the word "SLATE", and sections for "Tables", "Triggers", and "Database Scheme". The main area has a title "Tables" and a section titled "COUNTRIES". It contains a table with columns: column, nullable, type, descr, and fkey. The table has three rows: COUNTRY_ID (nullable N, type CHAR, descr: Primary key of countries table), COUNTRY_NAME (nullable Y, type VARCHAR2, descr: Country name), and REGION_ID (nullable Y, type NUMBER, descr: 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 a single row: DEPARTMENT_ID (nullable N, type NUMBER, descr: 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 DBDoc generates markdown out 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 your site, please, refer for instructions in Slate backend documentation.

Documenting PostgreSQL Database

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

Installing prerequisites

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.

Install PostgreSQL and its Python connector.

```
$ pip3 install psycopg2-binary
```

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](#).

Finally, [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 +     dbms: postgres
9 +     host: localhost
10 +    port: 5432
11 +    dbname: postgres
12 +    user: postgres
13 +    password: postgres
14 +   - plantuml
15 +
```

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 platnum.jar in preprocessor settings like this:

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

Finally, we need to point Foliant the place in the Markdown source files where the generated documentation should be inserted. 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 +
```

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 a dark-themed documentation interface for a database schema named 'SLATE'. On the left, there's a sidebar with a logo and a placeholder for a company logo. Below the logo, the word 'SLATE' is displayed in white. Underneath 'SLATE', it says 'REPLACE THIS WITH YOUR LOGO IN SOURCE/IMAGES/LOGO.PNG'. The sidebar has a 'Tables' section with a blue header, listing tables: COUNTRIES, DEPARTMENTS, EMPLOYEES, JOBS, JOB_HISTORY, LOCATIONS, and REGIONS. It also lists 'Triggers' and 'Database Schema'. The main content area is titled 'Tables' and contains two sections: 'COUNTRIES' and 'DEPARTMENTS'. The 'COUNTRIES' section includes a table description: 'country table. Contains 25 rows. References with locations table.' Below this is a table with columns: column, nullable, type, descr, and fkey. The first row shows COUNTRY_ID (nullable N, type CHAR) as the primary key for the countries table. The second row shows COUNTRY_NAME (nullable Y, type VARCHAR2) as the country name. The third row shows REGION_ID (nullable Y, type NUMBER) as the region ID, which is a foreign key to the region_id column in the departments table. The 'DEPARTMENTS' section includes a table description: 'Departments table that shows details of departments where employees work. Contains 27 rows; references with locations, employees, and job_history tables.' Below this is a table with columns: column, nullable, type, descr, and fkey. The first row shows DEPARTMENT_ID (nullable N, type NUMBER) as the primary key for the departments table.

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 DBDoc generates markdown out 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 your site, please, refer for instructions in Slate backend documentation.

Creating a Preprocessor

Introduction

Creating preprocessors for Foliant is quite straightforward because they are essentially just Python scripts wrapped in a `Preprocessor` class, which is provided by Foliant core. In this tutorial, we will go through all steps of creating a new preprocessor.

The full source code of the preprocessor created in this tutorial can be found [here](#).

First of all, we need to decide what our preprocessor will do. Let's say you need a preprocessor that will generate some placeholder gibberish text for your documentation, somewhat like [Lorem Ipsum](#).

We need a way to tell Foliant to insert the placeholder into a specific part of our document. The Foliant way of doing that is using an XML-tag like the following.

```
<gibberish></gibberish>
```

After the preprocessor is applied, this tag should be transformed into some placeholder text.

```
Hiteap zoiouxwaf jyrcaay yty xuzuapo eyuigouu. Ysseotaeq  
ytuiio qqyy yehiiy koyiyoky uul. Pan osfu zoiz oy ikcya  
tcsxecy qxiiyo. Gryxeye ogeelaee atprwm mjiroy eigyyoov nx qe  
tayoiaud jodmaofue yvo ieyuuny rq eaowu. Jyqnr aej elqj  
wuytjcae oy igy ak.
```

We would also want to specify the size of the generated text, so our tag should accept the `size` parameter which will define the number of generated sentences:

```
<gibberish size="15"></gibberish>
```

The tutorial is split into three stages:

1. Writing the gibberish generator,
2. Wrapping it in a Preprocessor class,
3. Installing and testing the preprocessor.

So let's get started!

Next: Creating the Gibberish Generator

Creating the Gibberish Generator

There are already several Python packages present on PyPi which generate placeholder texts like [lorem ipsum](#) but we won't deprive ourselves of the fun of creating our own.

Let's define some requirements:

- The generated text should consist of sentences that start with a capital letter and end with a dot.
- There should be a way of controlling the size of the sentence and the number of sentences in the resulting text.
- The words in the text should have at least a slight resemblance with real language words.

The last requirement is a bit tricky: we don't want words like `q` or `zxd` in our text, or at least not too many of those, so that the text looks a bit more real. So what we will

do is to create a simple `gen_word` function which will generate a word with a random number of letters, but the letters will be picked in a more controlled way by another function called `pick_letter`:

```
1 from random import randint
2
3 def gen_word():
4     word_len = randint(2, 9) # [1]
5     return ''.join(pick_letter() for _ in range(word_len))
# [2]
```

1. We've restricted the length of the words to 2 to 9 letters so we could avoid too short and too long words.
2. The `pick_letter` function will be supplying us with random letters.

Now to the `pick_letter` function. To make the words look real we don't want this function to return too many of the letters q, w, x and z, which don't appear in the words often. We also want to get more vowels than consonants: `kiobe` looks more like a word than `lknsd`.

Here's one way to do it

```
1 from random import choice, randint
2
3 def pick_letter():
4     rare_letters = 'qwxz'
5     vowels = 'aeiouy'
6     consonants = 'cdfghjklmmpqrstv'
7
8     pick = random() # [1]
9     if pick > 0.9: # [2]
10        return choice(rare_letters) # [3]
11    elif pick > 0.25: # [4]
12        return choice(vowels) # [5]
13    else:
14        return choice(consonants) # [6]
```

1. Get a random float number from the `random` function.
2. Since `random` returns a float from 0.0 to 1.0, there's about a 10% chance of getting a float that is larger than 0.9.

3. In this case, we will randomly pick one of the rare letters: q, w, x, or z with the `choice` function.
4. The chance of getting a float between 0.25 and 0.9 is about 65%.
5. In this case, we will return a vowel.
6. Finally, with a chance of about 25%, we will be returning one of the remaining consonants.

Let's put it all together and test our `gen_word` function

```

1 >>> gen_word()
2 'eojuo'
3 >>> gen_word()
4 'soe'
5 >>> gen_word()
6 'qwiim'
7 >>> gen_word()
8 'itookao'
```

Oh my god, I think we've just created the Finnish language! Jokes aside, it seems that our words generator works fine.

Now we need to create functions for generating sentences and putting them together in a text.

```

1 def gen_sentence(num_words=7):  # [1]
2     words = (gen_word() for _ in range(num_words))  # [2]
3     return ' '.join(words).capitalize()  # [3]
```

1. The number of words in the sentence is determined by the `num_words` parameter with a sensible default of 7 words.
2. Creating a `generator` that will yield a new word a required number of times.
3. Joining the generated words into a single string, separated by spaces. We are also making the first word capitalized in our sentence.

A quick test to make sure it works

```

1 >>> gen_sentence()
2 'Oveecyyi tukzgoli zvo uqyi ujiffrl viivu odui'
3 >>> gen_sentence(3)
4 'Ioyieyug ie hkeepnyo'
```

And now to the whole text generator

```

1 def gen_text(num_sentences=10): # [1]
2     sizes = (randint(3, 12) for _ in range(num_sentences))
# [2]
3     sentences = (gen_sentence(size) for size in sizes) #
[3]
4     return '. '.join(sentences) + '.' # [4]

```

1. The text generator will accept one parameter `num_sentences` with a default of 10.
2. Creating a generator that will yield a number of words in each sentence a required number of times. We are limiting the sentence size here to 3 to 12 words.
3. Creating a generator that will yield a new sentence a required number of times.
4. Joining the generated sentences into a single string, separated by dots. We are also adding a dot at the end of the text.

Time for the final test!

```

1 >>> gen_text()
2 'Eeaidmt cznm aeoiemino ivjuyauq exieh aoioayif yavfkoa
tasojm xuz qizxiyum iyo fajo. Anuipcauo uac eunjtou oiy
hougqf tulztiawk qooulu eiewewaii. Lxi isoxuau ooovox
wtopuodu oom ougvoeyy ou calxja io reicye yaioyzx. Usmyuavq
you xioqeи iiu ateuyau yeroueut gucuifuth tiazkkgc. Oyqzuy
rnzouq ajiof qaxewxufo. Utiselorc qpoaoydoi kyvyiuao ofxaoiy
ueyaoi azdacy lieaiiy au vteccye. Lopgygsz efixuio gi
eyzeuxoa eea qwaycx impoetvy eoyijaum uoiighcq lyaxa xy. Yo
yazd oio yyn gvyifzaeo eyz iewueuqze yy yeavtx dqmdiy.
Agiiorixk yae tvmu eeeoe aqjy eqnsouejn. Szejaae yl vuoaewt
aujc nvkols auokud reaqopae.'
3 >>> gen_text(2)
4 'Uyayu xpriicoe usao yua duleekayi loqk iop saiy iuys
sciyaihs. Onacrtog ual iei nuuoaz gdgia yyoui.'

```

Our gibberish generator turned out quite decent. Now it's time to make it a Foliant preprocessor.

[Next: Formalizing the Preprocessor](#)

[Previous: Introduction](#)

Formalizing the Preprocessor

A Foliant preprocessor is a Python package of a certain structure. Here's a list of requirements for a package to be considered by Foliant a preprocessor:

1. After installing the preprocessor package should appear inside the Foliant package folder at a path `foliant preprocessors.your_preprocessor`.
2. It must be possible to import a class named `Preprocessor` from your package:

```
from foliant.preprocessor.your_preprocessor import  
Preprocessor
```

3. The `Preprocessor` class should (but is not required to) be a subclass of a `foliant.preprocessors.BasePreprocessor` class.
 - in any case the `Preprocessor` class must accept the same `__init__` arguments as the `BasePreprocessor` class.

4. The `Preprocessor` class must define at least the `apply` method.

Let's take our `gibberish` module which we've created in the previous chapter and make it work with Foliant. We will be adding the code into the same module.

According to the requirements above, we first need to create a proper directory structure. It should look like this:

```
1 $ tree  
2 .  
3   foliant  
4     └── preprocessors  
5       └── gibberish.py  
6  
7 2 directories, 1 file
```

This is the first requirement satisfied. Now to the rest of them.

We should define a `Preprocessor` class and ideally subclass it from `BasePreprocessor`.

Usually, we start out a new preprocessor from a template containing the boilerplate code. You can find the full template [here](#).

But to understand the boilerplate code you have to write it at least once, so let's start from scratch.

We start by defining the `Preprocessor` class.

```
1 from foliant.preprocessors.base import BasePreprocessor  
2  
3  
4 class Preprocessor(BasePreprocessor):
```

The `BasePreprocessor` parent class offers some useful attributes and methods, go ahead and take a look at its [source code](#).

Let's start writing the class by adding the `__init__` method and several class attributes.

```
1 class Preprocessor(BasePreprocessor):  
2     tags = ('gibberish', ) # [1]  
3     defaults = {'default_size': 10} # [2]  
4  
5     def __init__(self, *args, **kwargs):  
6         super().__init__(*args, **kwargs) # [3]  
7  
8         self.logger = self.logger.getChild('gibberish') #  
9         [4]  
10            self.logger.debug(f'Preprocessor initied: {self.  
11             __dict__}') # [5]
```

1. First, we define the tags which will be captured in the Markdown source. As we've decided in the beginning, we want to process tags that look like `<gibberish></gibberish>`, so our tag name is `gibberish`. We put that into the `tags` class attribute which must be a sequence. `tuple` or `list` are equally good choices.
2. We will allow the user to define the default size of the generated text in the preprocessor options. Here we provide the default value of `10` to this option, in case the user hasn't supplied it.
3. Running the parent's `__init__` method first. It will populate our class with useful attributes.
4. Using the `logger` attribute to set up a logger. This line embeds our preprocessor into the main log file under the name of `gibberish`.
5. Posting our first log message, which will contain all preprocessor's attributes for inspection.

Now let's write the `apply` method. As mentioned above, this method must be present in all preprocessors. This is the method that Foliant will call to apply the preprocessor.

Usually, it subsequently opens each file from the temporary directory and calls the main processing method to transform their content in the desired way. It's a good practice to start and end this method with log messages.

```
1     def apply(self):
2         self.logger.info('Applying preprocessor Gibberish')
3         for markdown_file_path in self.working_dir.rglob('*.
4             md'): # [1]
5             with open(markdown_file_path, encoding='utf8')
6                 as markdown_file:
7                     content = markdown_file.read() # [2]
8
9                     processed_content = self._process_tags(content)
# [3]
10
11                if processed_content: # [4]
12                    with open(markdown_file_path, 'w') as
13                        markdown_file:
14                            markdown_file.write(processed_content)
# [5]
15
16                self.logger.info('Preprocessor Gibberish applied')
```

1. Scan the temporary directory (the `working_dir` attribute, which is a `pathlib.Path` object, created for us by the parent class) and find all Markdown files in it.
2. Get the source content of each Markdown file.
3. Process the content with the `_process_tags` method which we are about to write next.
4. This step is important. We check if the main processing method actually returned any content. If the string is empty, it usually means that something went wrong. Foliant won't interrupt the build process if one of the preprocessors fails to run. We don't want to write empty or broken content into Markdown files, because other preprocessors still may run after ours even if ours failed.
5. If everything is OK, and our preprocessing function returned some content, we overwrite the original Markdown file with it.

The `apply` method defers the actual preprocessing work to the `_process_tags` method, so now let's write it.

```

1   def _process_tags(self, content):
2       def sub_tag(match): # [2]
3           tag_options = self.get_options(match.group('
options')) # [3]
4           default_size = self.options['default_size'] # [4]
5           size = tag_options.get('size', default_size) #
[5]
6           return gen_text(size) # [6]
7
8       return self.pattern.sub(sub_tag, content) # [1]

```

Note the order of the bullet points: we start with the last line of the code above:

1. The `pattern` is another attribute created by the base class. It is a RegEx pattern object which will capture the XML tags in the Markdown source. Remember the `tags` class attribute we've defined in the beginning? `pattern` will use it to capture the appropriate tags for our preprocessor. We use the `re.sub` method of the pattern which will replace our tag definitions (`<gibberish></gibberish>`) in the `content` with whatever string returns the `sub_tag` local function.
2. Next, we define the `sub_tag` local function. This function accepts one argument: the `Match` object which was captured by the pattern.
3. We use the handy `get_options` method of the base class, which takes the options string of the tag found in the source, and turns it into a dictionary of options. For example, if the captured tag was `<gibberish size="15"></gibberish>`, the options string is `size="15"`. It will be turned into `{'size': 15}` by the `get_options` method.
4. Getting the value of the `default_size` parameter from the preprocessor options. The options are stored in `self.options` dictionary by the base class. The dictionary is first prepopulated by values from the `defaults` attribute that we've defined earlier. According to the `defaults`, if the user hasn't stated any options, the `default_size` will have the value of `10`.
5. Getting the `size` option from the tag options. If options were not stated, we are using the `default_size` value as a fallback.
6. Finally, we are using the `gen_text` function from our gibberish generator, which we've written in the previous part of the tutorial. We are returning the string returned from the `gen_text` function as the result of our `sub_tag` local function. This is the text which will replace the `<gibberish></gibberish>` tag in the processed Markdown file.

And that's it! We have all the code required for the preprocessor to work. All is left to do is to make our package installable and test its work.

Next: Installing and Testing

Previous: Creating the Gibberish Generator

Installing and Testing

Right now our preprocessor folder looks like this.

```
1 $ tree
2 .
3   foliant
4     └── preprocessors
5       └── gibberish.py
6
7 2 directories, 1 file
```

To make it an installable Python package we need to add a `setup.py` file to the root folder.

Here's an [article on creating setup files](#) from the official docs. Usually, we just take one of the `setup.py`s from an existing preprocessor as a template or use this official Foliant [snippet](#).

Here's what your `setup.py` may look like.

```
1 from setuptools import setup
2
3
4 SHORT_DESCRIPTION = 'Gibberish preprocessor for Foliant.' #
5   [*]
6
7 try:
8     with open('README.md', encoding='utf8') as readme:
9         LONG_DESCRIPTION = readme.read()
10
11 except FileNotFoundError:
12     LONG_DESCRIPTION = SHORT_DESCRIPTION
```

```

13
14 setup(
15     name='foliantcontrib.gibberish',    # [*]
16     description=SHORT_DESCRIPTION,
17     long_description=LONG_DESCRIPTION,
18     long_description_content_type='text/markdown',
19     version='1.0.0',
20     author='Simon Garfunkel',    # [*]
21     author_email='simong@example.com',  # [*]
22     url='https://github.com/foliant-docs/foliantcontrib.
23     gibberish',    # [*]
24     packages=['foliant.preprocessors'],
25     license='MIT',
26     platforms='any',
27     install_requires=[
28         'foliant>=1.0.8',
29     ],
30     classifiers=[
31         "Development Status :: 5 - Production/Stable",
32         "Environment :: Console",
33         "Intended Audience :: Developers",
34         "License :: OSI Approved :: MIT License",
35         "Operating System :: OS Independent",
36         "Programming Language :: Python",
37         "Topic :: Documentation",
38         "Topic :: Utilities",
39     ]
)

```

Lines marked with asterisks you would probably want to change to suit your preprocessor. Also, note that we referred to the contents of the `README.md` as the full description of the package. It's a good time to add a `readme` for your preprocessor. Explain what your preprocessor does and what options it has. You may use one of the official preprocessors for possible `readme` structure.

Now the folder structure should look like this

```
1 $ tree
```

```
2 .|  
3 foliant|  
4   └── preprocessors|  
5     └── gibberish.py|—  
6 README.md|—  
7 setup.py  
8  
9 2 directories, 3 files
```

Time to test if the preprocessor actually works. First, install it by running this command inside the preprocessor folder.

```
$ pip3 install .
```

Create an empty Foliant project using the `init` command:

```
1 $ foliant init # creating the empty project  
2 Enter the project name: Gibberish Test  
3 Generating project... Done—————  
4  
5 Project "Gibberish Test" created in gibberish-test  
6 $ cd gibberish-test # entering the created project folder  
7 $ tree # inspecting the project file structure  
8 .|  
9   docker-compose.yml|—  
10  Dockerfile|—  
11  foliant.yml|—  
12  README.md|—  
13  requirements.txt|—  
14  src  
15    └── index.md  
16  
17 1 directory, 6 files
```

First, let's add our preprocessor to the `foliant.yml`.

```
1 title: Gibberish Test  
2  
3 + preprocessors:
```

```
4 +     - gibberish
5 +
6 chapters:
7     - index.md
```

Now let's edit the `index.md` and use the `<gibberish></gibberish>` tag a few times.

```
1 # Welcome to Gibberish Test
2
3 Here's some gibberish:
4
5 <gibberish></gibberish>
6
7 Here are just two sentences of gibberish:
8
9 <gibberish size="2"></gibberish>
```

Let's build our project into the `pre` target. This target doesn't create a PDF or a DOCX, it just returns the preprocessed Markdown text, which perfectly suits our testing needs.

```
1 $ foliant make pre
2 Parsing config... Done
3 Applying preprocessor gibberish... Done
4 Applying preprocessor _unescape... Done
5
6 Result: Gibberish_Test-2021-08-16.pre
```

Inspect the results

```
1 $ cat Gibberish_Test-2021-08-16.pre/index.md
2 # Welcome to Gibberish Test
3
4 Here's some gibberish:
5
6 Yxz izyuo sjo iir tewo qvqc etosaeueo iecaizaso aaeoeuo iyey
. Apavaiqfu eqaaa eecyo ioiiyuoa ah caou iets. Yooyofa
iiynnnda yiuqeqlq uizu yca. Pi iuld ixuaeqli ousogp yu
```

```
ushggxyq yiia uiuyjo. Ofoemct ciyfuup uufiy avkfeqa ehtjoj
ietwohoo xqgif. Iwohqoeao snf uozlw qeasoqzu gevuywxui ou
xypikyyqu on hrx. Ruagoisia ivga ovzho da oziazioic. Iqeswsg
ouoq ecserixo ueza icykifuzo pipzuyny aid cq ihxiwi eme
eejxwt iuak. Oui goido yduz eeyfahxil dyiya mezifeo iym
xuuvyyiy. Iii yucnyyyq eono qyqu uu ioo sqwcjuhip.
```

7

8 Here are just two sentences of gibberish:

9

```
10 Lof peuoy iiouy yyau qggedo evuoucaig. Pziqqsg ekiqepyu
laeiridyc.
```

Looks like everything worked fine. Now let's set the `default_size` parameter to check if preprocessor options work too. Edit the `foliant.yml`

```
1 title: Gibberish Test
2
3 preprocessors:
4 - - gibberish
5 + - gibberish:
6 +     default_size: 1
7
8 chapters:
9 - index.md
```

And run the build

```
1 $ foliant make pre
2 Parsing config... Done
3 Applying preprocessor gibberish... Done
4 Applying preprocessor _unescape... Done
5
6 Result: Gibberish_Test-2021-08-16.pre
```

Now the first `<gibberish></gibberish>` tag should be replaced with just one line of text. Let's check that

```
1 cat Gibberish_Test-2021-08-16.pre/index.md
2 # Welcome to Gibberish Test
```

```
3
4 Here's some gibberish:
5
6 Jy si zhwtyu acneec qeugeya ax qqofaiu ydyyyxz.
7
8 Here are just two sentences of gibberish:
9
10 Wnuhocx uqny ns. Iu ieuiaeia iogyjyfy kl eyyeex agayii aioaac
    yacjume.
```

Everything works as expected. Now you can add a `LICENSE` and a `changelog.md` to your preprocessor folder and publish it in GitHub and pypi, so that others could use your wonderful creation too!

The repository with full code of Gibberish preprocessor is available [here](#).

Summary

Now you know the basics of creating preprocessors for Foliant. But there's a lot more to learn! Study the code of different preprocessors created by our team to find out different approaches to solving techwriters' problems. Refer to the Developer's Reference for all helper functions, classes and their attributes available for building Foliant extensions.

When you get comfortable creating simple preprocessors you may find the [utils package](#) useful. It contains different tools which perform common tasks in preprocessors like [dealing with the chapters section](#) in `foliant.yml` or efficiently [combining options](#) from `foliant.yml` and XML tags. There's also a powerful [BasePreprocessorExt class](#) which encapsulates some boilerplate code for your preprocessors and offers advanced tools for warnings output and more.

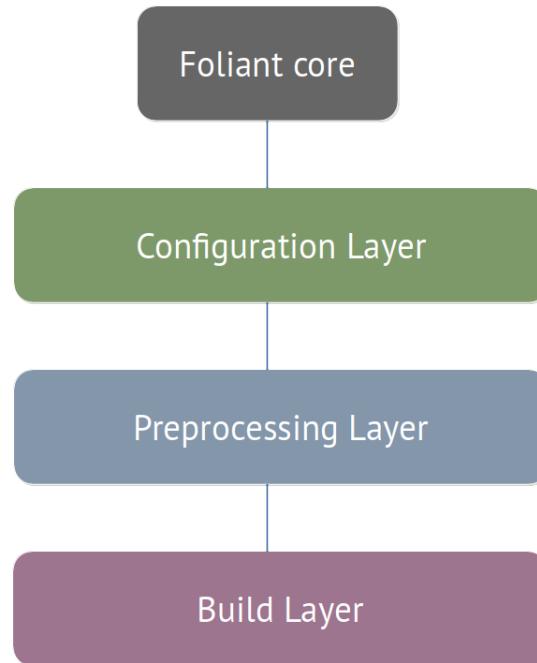
That's all for now. We wish you luck in extending Foliant! Send us a message if you want your preprocessor included in the official Foliant docs.

Previous: Formalizing the Preprocessor

Architecture And Basic Design Concepts

Overview

Foliant is an open-source application written in Python.

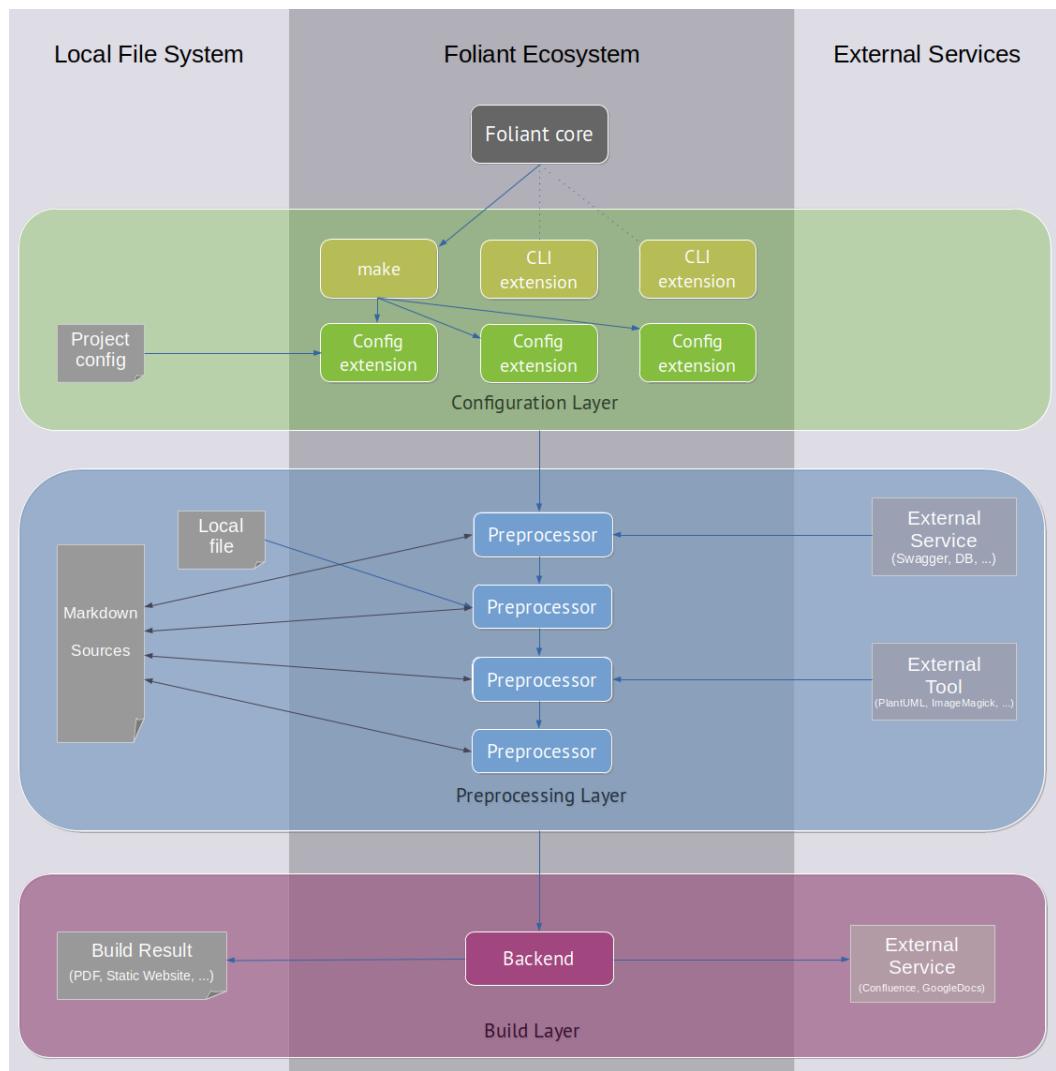


Foliant has a modular architecture. It consists of three layers:

- **Configuration Layer** reads Foliant project configuration file and CLI parameters supplied by the user;
- **Preprocessing Layer** adjusts the Markdown sources before the build;
- **Build Layer** produces the documentation in the final format.

Each layer is supervised by [Foliant Core](#). [Foliant Core](#) is a relatively compact and rarely updated Python package. It is a dispatcher which manages installed extensions according to the configuration.

Let's take a closer look at each layer of Foliant's architecture.



Configuration Layer

At the configuration Layer, Foliant processes the CLI-command and reads the command arguments, which were supplied by the user. The Main Foliant command is `make`, it is the command which builds the documentation project. But there are other commands which may do other things, for example, display the project's metadata (the `meta generate` command) or generate a new project file structure (the `init` command).

If `make` command was used, Foliant reads and processes the project configuration file. At this stage, all installed Configuration Extensions are applied to the project config.

Preprocessing Layer

At the preprocessing layer, Foliant runs the Preprocessor Pipeline, that was defined in the config. Each preprocessor is applied to the Markdown source in the specified order. Preprocessors may:

- call external services, for example, get data from a Swagger API website or an SQL Database;
- use local files, for example, templates for text generation, or Markdown snippets for content reuse;
- call external tools, for example, PlantUML to generate diagrams from code, or ImageMagick to resize images.

Finally, the preprocessor may not use any external services or local files, but do the processing itself. For example, perform auto-replace in the text or generate a glossary for the terms in the project.

Build Layer

At the build layer, Foliant runs the Backend, which was specified in the `make` command parameters. The Backend produces the documentation in the final format. It may be a local file like PDF or a directory with static website contents, or the Backend may upload the result to an external service like Confluence.

Foliant Extensions

As was mentioned above, the main Foliant package is Foliant Core. But Foliant Core itself does not build documentation projects, instead it delegates this job to extensions. The Core package also defines base classes for all types of extensions.

There are 4 types of base Foliant extensions.

- **CLI extensions** extend Foliant's command-line interface and provide additional actions that may be called from the command line. This is always the topmost component of any Foliant's action. `foliant make` is in fact a CLI extension that builds projects.

- **Config extensions** allow to customize the project configuration parsing, add custom YAML tags and new configuration options. For example, MultiProject extension adds a YAML tag `!from` which allows to include multiple nested Foliant projects into a single parent project.
- **Preprocessors** are modules which apply various transformations to the source Markdown content before passing it to a backend. The transformations include:
 - replacing parts of content according to specific rules;
 - rendering diagrams and schemes from source code;
 - embedding content from external files;
 - getting data for your documentation project from external services, e.g. remote Git repositories, Swagger, Testrail, Figma, Sympli, SQL Databases etc.;
 - setting high-level semantic relations between different parts of content to provide smart cross-target links, or restructure single-source documentation automatically and context-dependently;
 - running arbitrary external commands.

Each Foliant project may use any number of preprocessors. Preprocessors are applied sequentially, one after another. The same preprocessor may appear more than once in the pipeline.

- **Backends** build the project’s Markdown content into final formats which we call targets, e.g. PDF files or static sites. Backends may call third-party software to produce the final documentation or upload your content to an external service, e.g. Confluence. A single backend may generate multiple targets. Different backends may build the same target. For example, a static site (the `site` target) can be built with 3 official backends: MkDocs, Slate, and Aglio. If several of them are installed, user may specify the certain backend in the `foliant make` command or it will be asked interactively.

Project Build Process

The project build process is operated by Foliant CLI extension called make, which is a part of Foliant Core package.

The steps of the build process

1. User calls a `make` command specifying the backend and the target he wants to build, for example:

```
$ foliant make site --with mkdocs
```

In this example, the target is `site` and the backend is `mkdocs`. `--with` argument is optional, `make` will assume the backend or ask for user input if there are several options for the target.

2. `make` launches the project build in the following stages:
 1. **Configuration parsing.** The project configuration file (`foliant.yml` by default) is processed by each installed Config extension and saved into the internal context.
 2. **Copying sources.** The `src` folder which holds Markdown source files of the project is copied into a temporary folder (`__folianttmp__` by default). Pre-processors will only affect the copies, leaving the sources intact.
 3. **Preprocessing.** Each preprocessor defined in the project configuration file is subsequently applied to the temporary folder with copies of Markdown sources. The preprocessors run in an order in which they are specified in the `processors` list, but each backend may implicitly add specific preprocessors to the beginning or the end of this list.
 4. **Producing output format.** The chosen backend takes the Markdown files from the temporary folder and converts them into the target format.
 5. **Removing temporary files.** If `make` wasn't run with `--keep-tmp | -k` argument, the temporary folder with preprocessed Markdown sources is removed from the project dir.

Project Configuration

Configuration for Foliant is kept in a [YAML](#) file in the project root. The default filename is `foliant.yml` but you can pick a different name by specifying the `--config` option:

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

Config 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 and UnescapeCode). 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`, the output PDF will be saved into `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.
Default: `src`.

tmp_dir (string) Name of the directory where the intermediate files will be stored during preprocessor pipeline execution. Default: `__folianttmp__`.

chapters

(list)

`chapters` is a list of paths to the Markdown sources which you want to be used in the project. The paths are specified 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 nested with mappings and sublists. These complex structures may be treated differently by different backends: some may ignore nesting, some may use it 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    # list item  
3     - definitions.md    # list item  
4     - How To Use This Tutorial: tutorial_help.md    # mapping  
      with one element  
5     - Creating Documentation With Foliант:    # mapping with  
      nested list  
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 simple list items, the 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 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 were building a site with MkDocs backend, mappings would 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 Foliант:  
2         - prerequisites.md  
3         - Preparing Config:  
4             - root_options.md  
5             - chapters.md  
6             - preprocessors.md  
7             - backend_config.md
```

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

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: # options are adjusted
3         macros:
4             ref: <if backends="pandoc">{pandoc}</if><if
backends="mkdocs">{mkdocs}</if>
5         - flags # all options are set to defaults
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 a 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 key being the preprocessor name, and value – another mapping, with preprocessor options. (`macros`, `plantuml`, and `graphviz` in the example above).

Refer to each preprocessor’s respective docs for details on which options they have and how to set them.

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

The order matters

The order, in which the preprocessors are defined in the list, is the order they are run during the 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 preprocessor 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 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 keep your `src` dir tidy and only put there files that 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    mkdocs:
12        use_title: true
13        use_chapters: true
14        use_headings: true
15    mkdocs.yml:
16        repo_name: foliant-docs/docs
17        theme:
18            name: material
19            custom_dir: !path ./theme/

```

Unlike `preprocessors` section, `backend_config` is not a list but a mapping. Hence, 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 above.

`!include`

The `!include` modifier allows inserting 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 that is sitting in the project root, no matter where inside the src dir your current file is:

```
1 Here are the contents of this 'projects 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 the current file. But if you want to make things more explicit, you may add the `!rel_path` tag, which ensures that the path the preprocessor will get, will be relative to the current file:

```
1 Here are the contents 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 the project root.

!env

The `!env` modifier allows you to access environment variables in the config, as well as in tag options.

It is useful if you don't want to keep credentials in your config files, for example:

```
1 # foliant.yml  
2  
3 preprocessors:  
4     dbdoc:  
5         host: localhost  
6         user: admin  
7         password: !env DBA_PASSWORD
```

Now to build this project add the variable to your command:

```
DBA_PASSWORD=WQHsaio901SY foliant make pdf
```

Or, if you are using docker:

```
docker-compose run --rm -e DBA_PASSWORD=WQHsaio901SY foliant  
make pdf
```

Debugging Builds

Building simple documentation projects with Foliant is usually straightforward. But Foliant is a powerful, customizable, and very flexible tool, capable of turning your most complex ideas into beautiful documents. If you understand exactly what you want to achieve, you can formalize it at the project config level, and Foliant will perform your task efficiently and precisely.

But sometimes it is difficult to configure all preprocessors and backends properly in one go. Some settings are pretty subtle and some preprocessors are quite complicated. The order of applying the preprocessors matters. Some preprocessors may work unexpectedly when paired with others. Fetching data from external sources may also become a bottleneck. The list goes on.

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

- detailed event logging in the debug mode;
- the pre backend which does nothing, i.e. just returns the preprocessed Markdown;
- an option to keep the temporary working directory for further analysis.

Notes on Docker Use

In practice, Foliant is more commonly used with Docker.

Here's a tip for debugging Foliant projects with docker.

It's useful to add one more service to your project's default `docker-compose.yml`. We will call it `bash` and it will run containers with an interactive shell:

```
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
```

Now you can run a container based on the project's image with an interactive shell. To open the 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 debugging approaches which we will discuss next are represented as native Foliant commands, but they are applicable for the Docker way too. Just start your commands with `docker-compose run --rm` or `docker-compose run --user="$(id -u):$(id -g)" --rm` to run them within Docker containers.

Logging

The `foliant make ...` command runs Foliant in the regular logging mode. In this mode, Foliant and its extensions will only log events with levels of `critical`, `error`, and `warning`. Note that some preprocessors may generate a lot of specific warnings which may or may not indicate that something went wrong. These messages are usually worth studying anyway though.

The new log file is created for each build, unless there were no errors and warnings. The logs are stored by default in the project root under the name `<unix timestamp of the build>.log`, for example, `1628582527.log`. With such a naming convention the log file for the latest build will always be last in alphabetical order. The location of the log files may be customized by the `--logs | -l` command-line option.

Debugging Mode

Foliant provides the `--debug` or `-d` command-line option which enables the debugging mode. In this mode, Foliant and its extensions log not just events with levels `critical`, `error` and `warning`, but also events with levels `debug` and `info`. The amount of information you will get from such events depends on the implemen-

tation of a particular extension. Complex preprocessors like Includes usually log their actions in great detail. The messages of the `info` level are usually informative: they may mark the beginning or end of some preprocessor's work, for example. The messages of the `debug` level generally show the status of atomic operations, for example, reading data from a certain file. These messages often contain the values of the variables which are important in the current context: paths to files, external commands that are called, etc. But to make sense of these values prepare to get your hands dirty, or, in other words, read and understand the code of the corresponding extension.

Here's an example of a command that tells Foliant to build PDF with Pandoc in debug mode:

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

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

- date and time of the event registration;
- context (module name) in which the event was registered;
- 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 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 (Foliant Core).

The context is hierarchical. The following record represents an event that is registered in the Includes preprocessor which was implicitly called by the Flatten preprocessor, which was implicitly called during project build by Pandoc backend.

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

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 run it directly in an interactive shell and study the results.

Detailed logging in debug mode allows you to quickly localize problems zooming in from Foliant itself to a specific Foliant extension, a specific Markdown source file, or a specific line of code. This takes effort but with practice allows one to solve complex problems in minimal time.

Debugging extensions

Each Foliant backend takes preprocessed Markdown content and passes it to an external command (see [Architecture And Basic Design Concepts](#)). For debugging backends it's essential to see the content which the backend actually gets.

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

The `pre` backend

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

The `pre` target is essentially the content that comes after all preprocessors are applied, but before any backend (other than `pre`) is called.

Determining the cause of the problem

`pre` backend is convenient to determine the stage of a build which causes problems:

- configuration stage (reading the configuration file),
- preprocessing stage (transforming the Markdown content with extensions)
- or backend stage (producing the output format).

If you build a `pre` target and the results seem fine, then there is a problem with your backend and you have to debug that. You may start with the `keep-tmp` which we will discuss next.

If the problem persists in the results, produced by `pre`, then it's one of the preprocessors causing trouble or the configuration parser not working properly. In this case, it's a nice idea to stick with the `pre` target during your experiments so you won't need to wait each time for the backend to complete producing the target while you are debugging the build.

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

```
$ foliant make pre
```

Keeping the project sources

In addition to the `pre` backend, Foliant Core supports the `--keep_tmp` or `-k` command-line option. By default, the temporary working directory (`__folianttmp__`) is removed after the project build. But if the `--keep-tmp` or `-k` option is specified, the temporary working directory will stay in the project root after build.

This directory will contain the files that are modified by all preprocessors and the chosen backend.

If you have determined that the backend causes issues, `pre` won't help you anymore. Run the build with `-k` argument and study the working dir. If it seems fine, then the problem may be with the command that the backend runs to convert Markdown to a target format. Time to study logs!

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

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

Killing Two Birds With One Stone

Now you know what debugging facilities are provided by Foliant. But we strongly recommend you 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.

Metadata

User's guide

Metadata in Foliant allows you to assign additional properties to the [chapters](#) (Markdown files) and [sections](#) (parts of a Markdown file) of your project. These properties will be present in the Markdown sources but won't be directly rendered in the built documents. It is up to extensions to make use of these properties and alter your document in the desired way.

For instance, [Confluence](#) backend uses metadata to upload specific parts of your project into separate Confluence articles. [AltStructure](#) config extension uses metadata to rearrange the chapters of your project in the final build. [TemplateParser](#) preprocessor can access the metadata and generate chunks of text using the properties defined in it.

The `foliantcontrib.meta` package is required for metadata to work, but you won't need to install it directly. Every extension which uses metadata will install it automatically.

Syntax

There are two ways to define metadata:

- In a [YAML Front Matter](#) – to define metadata for a whole chapter,
- Using the `<meta></meta>` tag to define metadata for a section, as well as for the chapter.

YAML Front Matter

YAML Front Matter (or YFM for short) must be defined at the very beginning of a Markdown file. Properties in the YFM are applied to the whole chapter.

```
1 ---
2 author: John Smith
3 revision_date: 17 August 2021
4 ---
```

In this example we've defined two properties: `author` and `revision_date` for one chapter.

Meta tag

Meta tags may add properties to smaller chunks of a Markdown file, as well as the whole chapter. If the meta tag is specified at the very beginning of the file, it acts similarly to the YAML Front Matter, e.g. is applied to the whole chapter. To add properties to a smaller chunk of a Markdown file, specify the tag under a heading. The metadata will be applied to the text under the heading and all nested headings.

```
1 # Specification
2
3 <meta author="John Smith" revision_date="17 August 2021"></
  meta>
4
5 Lorem, ipsum dolor sit amet consectetur adipisicing elit.
  Aliquid neque, in, necessitatibus maxime repudiandae cum.
6
7 ## Additional notes
8
9 Lorem ipsum, dolor, sit amet consectetur adipisicing elit.
  Incidunt pariatur, vel voluptatum exercitationem quae
  cupiditate!
```

In this example both `Specification` and `Additional notes` have the `author` and `revision_date` properties.

Sections

Section is a part of a Markdown file with defined metadata. Section begins with a Markdown heading (`## Heading`) and ends before the next heading of the same or higher level (`## Another heading` or `# Another heading`). A part of a Markdown document is only considered a section if the meta tag is defined in it, with one exception: the main section.

The main section is defined implicitly for every chapter of your project, even if there's no meta tag or YFM in it. In other words, if the Markdown file is specified in the `chapters` in `foliant.yml`, it will appear in the meta registry, with or without meta properties.

Here's an illustration of meta sections in a chapter:

```

---  

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, 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 similique repudiandae non corporis omnis eum doloribus magnam, officia temporibus.


```

Special fields

Most meta properties don't mean anything if no extension is using them. The only exception right now is the `id` property. It is the identifier of a section.

IDs are used to distinguish meta sections in the project. They must be unique inside the project. By default IDs are generated by the Meta engine implicitly, but you may override them by defining the `id` property in the section's metadata. Just make sure that it is unique.

The Meta registry

All extensions that work with metadata have access to the [Meta registry](#). It is a hierarchical mapping of all sections in the project with all meta properties defined for each section.

To take a look at the Meta registry in your project run the `meta generate` command

```
1 $ foliant meta generate
2 Generating metadata... Done
3
4 Result: meta.yml
```

The registry is saved into the `meta.yml` file.

Additional info

Metadata works only for files, mentioned in the `chapters` section in `foliant.yml`. All other files in `src` dir are ignored and won't appear in the Meta registry.

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 extensions. You can define fields with special meaning for your tools and process sections based on the values in 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 returns the Meta registry in a `Meta` object, which gives access to all sections and meta-fields in the project.

The required parameter for `load_meta` 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 project's 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 an 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 an absolute path).

get_by_id(self, id_: str) -> Section

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

chapters

This property 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` (e.g. '`chapter.md`').

filename

Chapter's filepath string (e.g. '`src/chapter.md`').

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 a main section or not.

important properties

id

Holds section's ID.

title

Section's title.

chapter

Holds a reference to the section's `Chapter` object.

parent

Holds a reference to the section's parent section (`Section` object). Main sections have `None` in this property.

children

Holds list of section's children (`Section` objects) in proper order.

data

Holds a dictionary with meta properties and their values, defined in the `<meta>` tag (or the 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 the level `3`.

start and **end**

Section's offsets from the beginning of the Markdown file.

filename

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

Developer's Reference

The power of Foliant is in its extensions. Foliant's ecosystem consists of many beautiful tools for technical writers, but there is still a lot to be done. You are welcome to contribute to Foliant and its extensions.

This article contains the reference of the main classes and functions available in Foliant Core. As an extension developer, you will be using them to write your own preprocessors, backends, CLI- and config-extensions.

If you are new to extending Foliant, we suggest you to take a look at the [Creating a Preprocessor](#) tutorial first.

Official Foliant extensions live in Git repositories inside the [foliant-docs](#) GitHub group. Check out their source code to find out different approaches to solving techwriters' problems.

The repo of Foliant Core is called [foliant](#). The names of Foliant extensions' repositories start with the `foliantcontrib.` prefix. The repo of this documentation project is called [docs](#).

Core Modules

Core modules live in the [foliant](#) GitHub repository. Foliant Core itself does not build documentation projects, this job is delegated to extensions. But it defines the base classes for all types of extensions. Each base class offers useful attributes and methods which are described later in this article. For more info on how Foliant works check the [Architecture And Basic Design Concepts](#) article.

This section lists all modules in the Foliant Core package.

- `foliant`:
- `backends`:
 - `base` – defines the base class for all backends;
 - `pre` – simplest backend that returns Markdown content processed by specified preprocessors as a build result;
- `preprocessors`:
 - `base` – defines the base class for all preprocessors;
 - `_unescape` – simple preprocessor that escapes pseudo-XML tags (which are normally recognized by other preprocessors as control sequences) in code examples. If you want an opening tag to be ignored by any prepro-

cessor, precede this tag with the `<` character. The `_unescape` preprocessor removes these characters before build. Instead of the `_unescape` preprocessor, you may 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 for calling Foliant. Nested modules:
 - `base` – defines the base class for all CLI extensions;
 - `make` – provides the main Foliant’s `make` command;
- `config`:
 - `base` – defines the base class for all config extensions;
 - `include` – resolves the `!include` YAML tag that allows to include the content of additional YAML-files in Foliant config. More info in the Project Configuration article;
 - `path` – resolves the `!path`, `!project_path` and `!rep_path` YAML tags. These tags are useful for specifying file paths in Foliant config or tag attributes. More info in the Project Configuration article;
 - `utils` – defines basic methods that may be used in different types of extensions.

The `make()` Method Arguments

The `make()` method is defined in the `foliant.cli.make` module. This method is called when the user runs `foliant make ...` command. For more info on how `make` command works check the Project Build Process article.

The `make()` method accepts a number of arguments; some of them are then passed to the backends and preprocessors in the build [context](#):

- `target` (string) – required resulting target of the current build;
- `backend` (string, defaults to an empty string) – the name of the backend that is used for the current build;
- `project_path` (path, defaults to the current directory path) – the path of top-level, “root” directory of the current Foliant project;
- `config_file_name` (string, defaults to `foliant.yml`) – the file name of the Foliant project’s config;
- `quiet` (boolean, default to `False`) – a flag that prohibits writing to `STDOUT`;
- `keep_tmp` (boolean, defaults to `False`) – a flag that tells Foliant and its extensions to preserve the temporary working directory, which is used during the build;

- `debug` (boolean, defaults to `False`) – a 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 extension.

- `BaseBackend()` is defined in the `foliant.backends.base` module. It is the base class for all backends. Each newly developed backend should:
 - be a module or a package `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 `Backend` class.
- `BasePreprocessor()` is defined in the `foliant.preprocessors.base` module. It is the base class for all preprocessors. Each newly developed preprocessor should:
 - be a module or a package `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()` is defined in the `foliant.cli.base` module. It is the base class for all CLI extensions. Each newly developed CLI extension should:
 - be a module or a package `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()` is defined in the `foliant.config.base` module. It is the base class for all config extensions. Each newly developed config extension should:
 - be a module or a package `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()`.

The BaseBackend() Attributes

- Class attributes:
 - `targets` (tuple of strings) – names of the targets that the backend can build;
 - `required_preprocessors_before` (tuple of strings) – names of the preprocessors that should be applied before all other preprocessors when this backend is used;
 - `required_preprocessors_after` (tuple of strings) – names of preprocessors that should be applied after all other preprocessors when this backend is used;
- instance variables:
 - `context` – a dictionary that contains the build context:
 - `project_path` (path) – path to the currently built Foliant project;
 - `config` (dictionary) – full config of the currently built Foliant project;
 - `target` (string) – the name of the resulting target;
 - `backend` (string) – the name of the backend that is used in the current build;
 - `config` – full config of the currently built Foliant project. The same as `context['config']`;
 - `project_path` – path to the currently built Foliant project. The same as `context['project_path']`;
 - `working_dir` (path) – the path to the temporary working directory that is used during the build. It is defined as `self.project_path / self.config['tmp_dir']`;
 - `logger` – the [Logger](#) instance of the current build;
 - `quiet` (boolean) – if `True`, the backend should not write anything to stdout;
 - `debug` (boolean) – if `True`, the backend should log the messages of `info` and `debug` levels.

The BasePreprocessor() Attributes

- Class attributes:
 - `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:
 - `context` – a dictionary that contains the build context:

- `project_path` (path) – path to the currently built Foliant project;
- `config` (dictionary) – full config of the currently built Foliant project;
- `target` (string) – the name of the resulting target;
- `backend` (string) – the name of the backend that is used in the current build;
- `config` – full config of the currently built Foliant project. The same as `self.context['config']`;
- `project_path` – path to the currently built Foliant project. The same as `self.context['project_path']`;
- `working_dir` (path) – the path to the temporary working directory that is used during the build. It is defined as `self.project_path / self.config['tmp_dir']`;
- `logger` – the `Logger` instance of the current build;
- `quiet` (boolean) – if `True`, the backend should not write anything to `stdout`;
- `debug` (boolean) – if `True`, the backend should log the messages of `info` and `debug` levels.
- `options` (dictionary) – the preprocessor’s options. Is defined as `{**self.defaults, **options}`, where `options` is the data that is read from the preprocessor’s config in `foliant.yml`;
- `pattern` – the regular expression `pattern` that is used to get components of a pseudo-XML tag in an easy way. Defined if `self.tags` is not empty. Provides the RegEx groups with the following names:
 - `tag` – captured tag name;
 - `options` – captured tag attributes (options) as a string; this string may be converted into a dictionary by using the `self.get_options()` method, which is provided by the base class;
 - `body` – captured tag body, i.e. the content between the opening and closing tags.

BaseCli() Attributes

- Instance attributes:
 - `logger` – the `Logger` instance of the current build.

BaseConfig() Attributes

- Instance attributes:
 - `project_path` (path) – the path to the currently built Foliant project;

- `config_path` (path) – the path to the config file of the currently built Foliant project;
- `logger` – the [Logger](#) instance of the current build;
- `quiet` (boolean) – if `True`, the config extension should not write anything to `stdout`.

Backends

Aglio

pypi v1.0.0

GitHub v1.0.0

Aglio Backend for Foliant

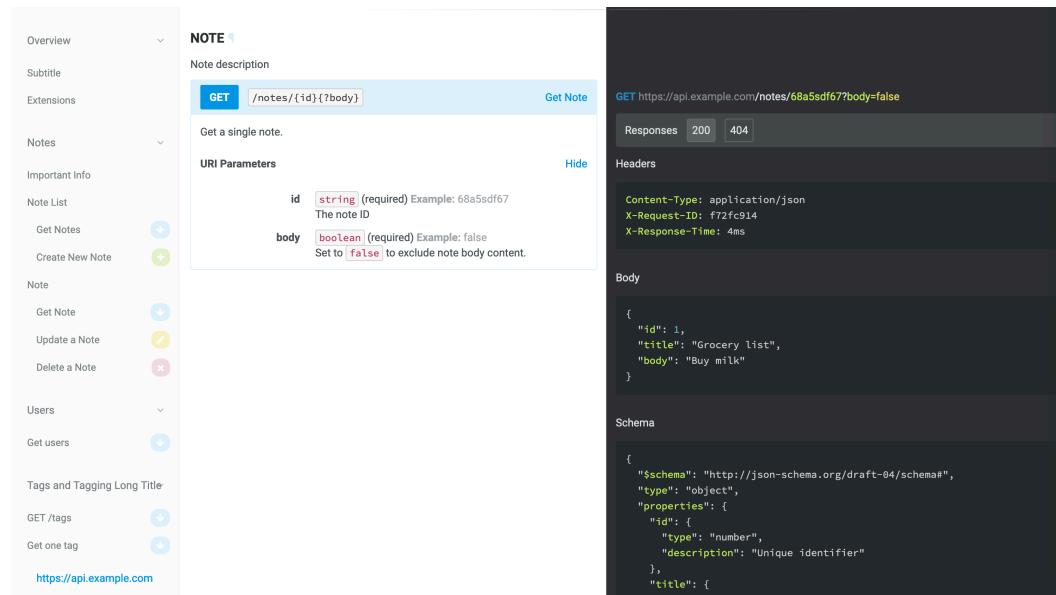


Figure 13. Static site built with Aglio backend

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 v0.6.20

GitHub v0.6.20

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 objects A and B.
 - Code Block:** Displays Python code for a class named `QueryBase` with methods for initializing a database connection and resolving filters.
 - Nested Lists:** A section titled "nested lists" containing two items: "1. First item." and "2. Second item."

Figure 14. 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  verify_ssl: true
20  attachments:
21    - license.txt
22    - project.pdf
23  codeblocks:
24  ...

```

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 binary (Pandoc is used to convert Markdown into HTML).

verify_ssl If `false`, SSL verification will be turned off. Sometimes when dealing with Confluence servers in Intranets it's easier to turn this option off rather than fight with admins. Not recommended to turn off for public servers in production. Default: `true`

attachments List of files (relative to project root) which should be attached to the Confluence page.

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
9 <meta
10     confluence=""
11         title: 'System components'
12         space_key: '~user'
13     ">
14 </meta>
15 ...
16
17 Lorem ipsum dolor sit amet, consectetur adipisicing elit.
18 Vel, atque!
19
20 ...
```

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 back-end 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.
  Odit dolorem nulla quam doloribus delectus voluptate.

2
3 <raw_confluence><ac:structured-macro ac:macro-id="1" ac:name="toc" ac:schema-version="1"/></raw_confluence>
4
5 Lorem ipsum dolor sit amet, consectetur adipisicing elit.
  Officiis, laboriosam cumque soluta sequi blanditiis,
  voluptatibus quaerat similiique nihil debitis repellendus.

```

In version 0.6.15 we've added an experimental feature of automatically escaping `<ac:...></ac:...>` tags for you. So if you want to insert, say, an image with native Confluence tag `ac:image`, you don't have to wrap it in `raw_confluence` tag, but keep in mind following caveats:

- singleton `ac:...` tags are not supported, so `<ac:emoticon ac:name="cross" />` will not work, you will have to provide the closing tag: `<ac:emoticon ac:name="cross"></ac:emoticon>`,

- only `ac:...` tags are escaped right now, other Confluence tags like `ri:...` or `at:...` are left as is. If these tags appear inside `ac:...` tag, it's ok. If otherwise, `ac:...` tag appears inside `at:...` or `ri:...` tag, everything will break.

Attaching files

To attach an arbitrary file to Confluence page simply put path to this file in the `attachments` parameter in `foliant.yml` or in meta section.

This will just tell Foliant to attach this file to the page, but if you want to reference it in text, use the other approach:

Insert Confluence `ac:link` tag to attachment right inside your Markdown document and put local path to your file in the `ri:filename` parameter like this:

```

1 Presentation in PDF:
2
3 <ac:link>
4   <ri:attachment ri:filename="presentation.pdf"/>
5 </ac:link>
```

In this case Foliant will upload the `presentation.pdf` to the Confluence page and make a link to it in the text. The path in `ri:filename` parameter should be relative to current Markdown file, but you can use `!path`, `!project_path` modifiers to reference images relative to project root.

Advanced images

Confluence has an `ac:image` tag which allows you to transform and format your attached images:

- resize,
- set alignment,
- add borders,
- etc.

Since version `0.6.15` you have access to all these features. Now instead of plain Markdown-image syntax you can use native Confluence image syntax. Add an `ac:image` tag as if you were editing page source in Confluence interface and use local relative path to the image as if you were inserting Markdown-image.

For example, if you have an image defined like this:

```
![My image](img/picture.png)
```

and you want to resize it to 600px and align to center, replace it with following tag:

```
1 <ac:image ac:height="600" ac:align="center">
2     <ri:attachment ri:filename="img/picture.png" />
3 </ac:image>
```

As you noticed, you should put path to your image right inside the `ri:filename` param. This path should be relative to current Markdown file, but you can (since 0.6.16) use `!path`, `!project_path` modifiers to reference images relative to project root.

Here's [a link to Confluence docs](#) about `ac:image` tag and all possible options.

If you want to upload an external image, you can also use this approach, just insert that proper `ac:image` tag, no need for `raw_confluence`:

```
1 External image:
2
3
4 <ac:image>
5 <ri:url ri:value="http://confluence.atlassian.com/images/
 logo/confluence_48_trans.png" /></ac:image>
```

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.

linenowbers 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 several 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. Omit credentials in config

A slightly more secure way is to remove password or both login and password from config:

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

In this case Foliant will prompt for missing credentials during each build:

```
1 $ foliant make confluence
2 Parsing config... Done
3 Applying preprocessor confluence_final... Done
4 Making confluence...
5
6 !!! User input required !!!
7 Please input password for user:
8 $
```

3. Using environment variables

Foliant 1.0.12 can access environment variables inside config files with `!env` modifier.

```
1 backend_config:
2     confluence:
3         host: https://my_confluence_server.org
4         login: !env CONFLUENCE_USER
5         password: !env CONFLUENCE_PASS
```

Now you can add these variables into your command:

```
CONFLUENCE_USER=user CONFLUENCE_PASS=pass foliant make
confluence
```

Or, if you are using docker:

```
docker-compose run --rm -e CONFLUENCE_USER=user -e
CONFLUENCE_PASS=pass foliant make confluence
```

4. Using passfile

Finally, 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" under the "Foliant" project. The sidebar on the left contains links for "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 has a heading "MkDocs" and a paragraph explaining the backend's purpose. It includes three code snippets: one for "Installation" showing the command \$ pip install foliantcontrib.mkdocs; one for "Usage" showing the command \$ foliant make mkdocs -p my-project with a list of steps: ✓ Parsing config, ✓ Applying preprocessor mkdocs, ✓ Making mkdocs with MkDocs; and one for "Build a standalone website" showing the command \$ foliant make site -p my-project with a list of steps: ✓ Parsing config.

Figure 15. 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 v1.1.2](#)

[GitHub v1.1.2](#)

Pandoc Backend for Foliant

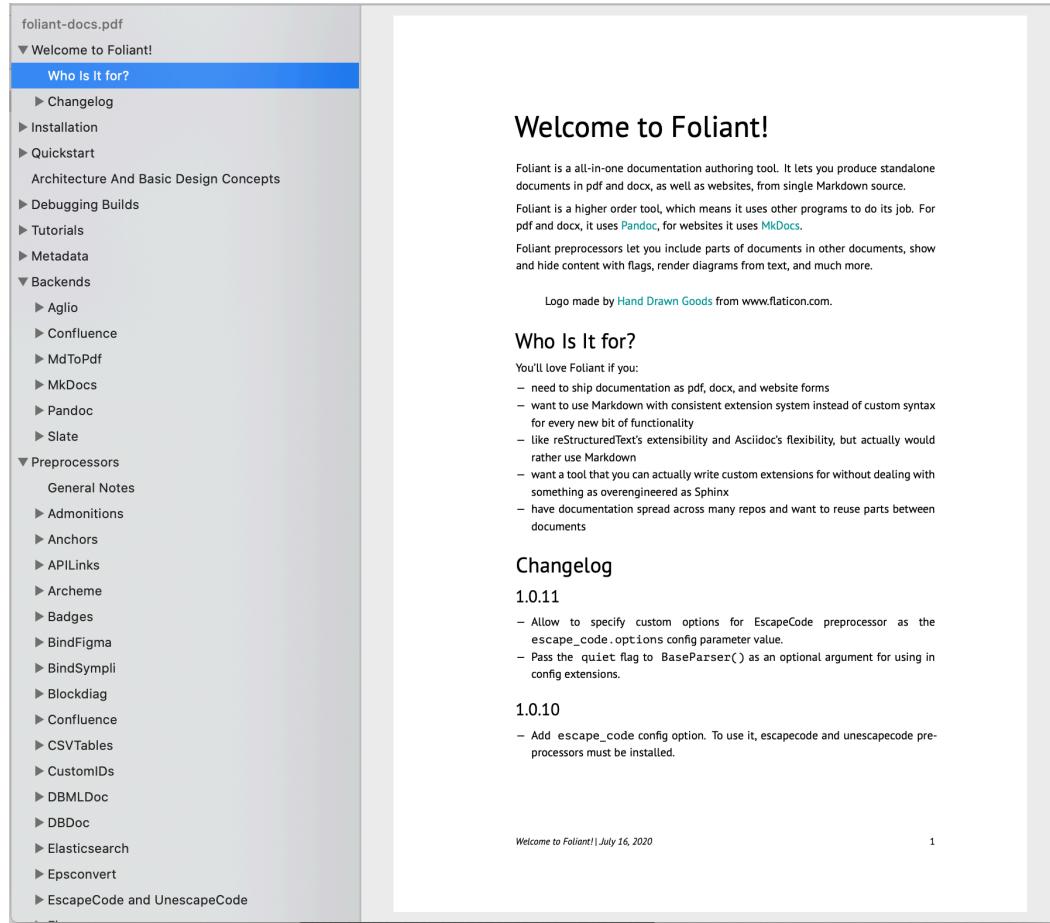


Figure 16. PDF built with Foliant

[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 adds `pdf`, `docx`, `odt`, `epub` 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... Done
3 Applying preprocessor flatten... Done
4 Making pdf with Pandoc... Done—————
5
6 Result: My_Project-2020-12-04.pdf
```

Build docx:

```
1 $ foliant make docx -p my-project
2 Parsing config... Done
3 Applying preprocessor flatten... Done
4 Making docx with Pandoc... Done—————
5
6 Result: My_Project-2020-12-04.docx
```

Build odt:

```
1 $ foliant make odt -p my-project
2 Parsing config... Done
3 Applying preprocessor flatten... Done
4 Making odt with Pandoc... Done—————
5
6 Result: My_Project-2020-12-04.odt
```

Build docx:

```
1 $ foliant make epub -p my-project
2 Parsing config... Done
3 Applying preprocessor flatten... Done
4 Making epub with Pandoc... Done—————
5
6 Result: My_Project-2020-12-04.epub
```

Build tex (mostly for pdf debugging):

```
1 $ foliant make tex -p my-project
2 Parsing config... Done
3 Applying preprocessor flatten... Done
4 Making docx with Pandoc... Done
5
6 Result: My_Project-2020-12-04.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     build_whole_project: true
5     template: !path template.tex
6     vars:
7       ...
8     meta:
9       ...
10    reference_docx: !path reference.docx
11    reference_odt: !path reference.odt
12    css: !path epub.css
13    params:
14      ...
15    filters:
16      ...
17    markdown_flavor: markdown
18    markdown_extensions:
19      ...
20    slug: My_Awesome_Custom_Slug
```

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

build_whole_project added in 1.1.0 If true, whole project will be built into a single flat document. Default: true.

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. They will be added to pandoc command as --variable key[=val].

meta is a mapping of document metadata properties and their values. They will be added to pandoc command as --metadata key[=val].

reference_docx is the path to the reference document to be used when building docx (see **-reference-doc** option info in the Pandoc [Options](#) documentation).

reference_odt is the path to the reference document to be used when building odt (see **-reference-doc** option info in the Pandoc [Options](#) documentation).

css is the path to the stylesheets to be used when building epub (see **-css** option info in the Pandoc [Options](#) documentation).

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: 2020
10    params:
11      pdf_engine: xelatex
12      listings: true
13      number_sections: true
14    markdown_extensions:
15      - simple_tables
16      - fenced_code_blocks
17      - strikeout

```

Build modes

Since 1.1.0 you can build parts of your project into separate PDFs, along with the main PDF of the whole project.

If the `build_whole_project` parameter of Pandoc backend config is `true`, the whole project will be built in to a flat document as usual. You can disable it by switching `build_whole_project` to `false`.

You can also build parts of your project into separate documents. To configure such behavior we will be adding [Metadata](#) to chapters or even smaller sections.

To build a chapter into a separate document, add the following `meta` tag to your chapter's source:

```

1 <meta
2   pandoc=""
3   vars:
4     toc: true
5     title: Our Awesome Product
6     second_title: Specifications
7     logo: templates/logo.png

```

```
8     year: 2020
9   "></meta>
10
11 # Specifications
12
13 size: 15
14 weight: 59
15 lifespan: 9
```

In the example above we have added a `meta` tag with `pandoc` field, in which we have overriden the `vars` mapping. The `pandoc` field is essential in this case. This is how backend determines that we want this chapter built separately. If you don't want to override any parameters, you can just add `pandoc="true"` field.

All parameters which are not overriden in the meta tag will be taken from main config `foliant.yml`.

Now, as the `pandoc` field is present in one of the meta tags in the project, Pandoc backend should build not one but two documents. Let's check if it's true:

```
1 $ foliant make pdf
2 Parsing config... Done
3 Applying preprocessor flatten... Done
4 Making pdf with Pandoc... Done—————
5
6 Result:
7 My_Project-2020-12-04.pdf
8 Specifications-2020-12-04.pdf
```

That's right, we've got the main PDF with whole project and another pdf, with just the `Specifications` chapter.

If you wish to build even smaller piece of the project into separate file, add meta tag under the heading which you want to build:

```
1 # Specifications
2
3 size: 15
4 weight: 59
5 lifespan: 9
6
```

```

7 ## Additional info
8
9 <meta
10    pandoc="
11      slug: additional
12      vars:
13        toc: true
14        title: Our Awesome Product
15        second_title: Additional info
16        logo: templates/logo.png
17        year: 2020
18    "></meta>
19
20 Lorem ipsum dolor sit amet consectetur adipisicing elit.
  Deleniti quos provident dolores eligendi nam quia sequi et
  tempore enim blanditiis, consequatur nostrum nulla dolor
  laborum quasi molestiae perspiciatis magni error consectetur
  nesciunt eaque veritatis voluptates! Cupiditate illum enim
  id recusandae assumenda excepturi odit tempore incidunt,
  amet soluta necessitatibus corrupti, aliquam.
```

In this example only the Additional info section will be built into a separate document. Notice that we've also given it its own slug.

Let's build again and look at the results:

```

1 $ foliant make pdf
2 Parsing config... Done
3 Applying preprocessor flatten... Done
4 Making pdf with Pandoc... Done—————
5
6 Result:
7 My_Project-2020-12-04.pdf
8 additional.pdf
```

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 [SSHк-  
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

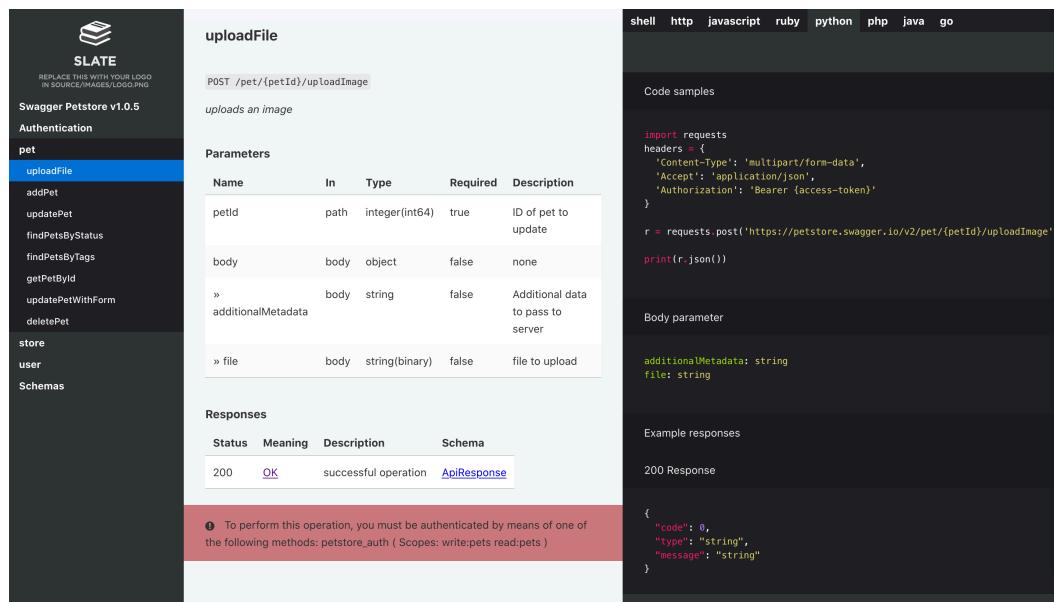


Figure 17. Static site built by Foliant and Slate backend

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 v1.0.1

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 v1.0.7](#)

[GitHub v1.0.7](#)

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  
5         anchors: True  
6         custom_ids: 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. Ignored when `tex` is `True`. Default: `''`

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`.

anchors If this options is `True`, anchors tag will be processed. Turn off if you only want to process custom IDs. Default: `True`

custom_ids Since version 1.0.5 preprocessor Anchors can also process Pandoc-style custom IDs. Set this option to `True` to do that. Default: `False`.

Usage

anchors

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.
2
3 [Go to the middle of the paragraph](#middle)
```

You can place anchors inside tables:

```
1 Name | Age | Weight
2 ---- | --- | -----
3 Max | 17 | 60
4 Jane | 98 | 12
5 John | 10 | 40
6 Katy | 54 | 54
7 Mike <anchor>Mike</anchor>| 22 | 299
8 Cinty| 25 | 42
9
10 ...
11
12 Something's wrong with Mike, [look](#Mike)!
```

custom IDs

Since version 1.0.5 preprocessor Anchors can also process [Pandoc-style custom heading identifiers](#) (previously you had to use [CustomIDs preprocessor](#) for that purpose). To use this function, turn on the `custom_ids` option in your foliant.yml:

```
1 preprocessors:  
2     anchors:  
3         ...  
4         custom_ids: True
```

Then add custom identifiers to your headings:

```
1 # My heading {#foo}  
2  
3 Lorem ipsum, dolor sit amet consectetur adipisicing elit.  
Omnis non vitae placeat sapiente reprehenderit officia.
```

After processing your text will look like this:

```
1 <span id="custom_id"></span>  
2  
3 # My heading  
4  
5 Lorem ipsum, dolor sit amet consectetur adipisicing elit.  
Omnis non vitae placeat sapiente reprehenderit officia.
```

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 you are building a flat document (e.g. PDF or docx with Pandoc), you will receive the warning even if anchor repeats in different md-files.

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.

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 change the `element` option for your anchors to have a vertical offset. Example config:

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

Or, even better, you can assign your anchor a class in `element` and add these rules to your custom mkdocs styles.

APIReferences

pypi v1.0.2

GitHub v1.0.2

APIReferences Preprocessor for Foliant

APIReferences is a successor of APILinks preprocessor with slightly changed configuration syntax and completely rewritten insides. APILinks is now deprecated, please use APIReferences instead.

Preprocessor replaces API references in markdown files with links to corresponding method description on the API documentation web-page.

What is it for?

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

It may be a [Swagger UI](#) website or just some static one-page site (like [Slate](#)).

If you have a site with API docs, you probably reference them from time in your other documents:

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

We thought, how cool it'd be if this fragment: '**POST /user/authenticate**' automatically transformed into a URL of this method's description on your API docs website:

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

That's exactly what APIReferences does.

How does it work?

The purpose of APIReferences is to convert references into links. In the example above ' POST /user/authenticate' is a reference, and [POST

`user/authenticate](http://example.com/api-docs/#post-user-authenticate)` is a Markdown link, the result of APIReferences' work.

The resulting link URL (`http://example.com/api-docs/#get-user-authenticate`) always consists of two parts: `{url}{anchor}`. `url` is static and is set in config, but `anchor` differs for each method. Open your API documentation website and look for HTML elements with `id` attribute near method description sections. When you add this `id` to the website's URL with number sign `#` (we call this combination an `anchor`), your browser scrolls the page to this exact element.

The tricky part is to determine which anchor should be added to the website's URL for each method. APIReferences offers several ways to do that, we call these ways modes (which are supplied in the `mode` parameter). It's up to you to choose the most suitable mode for your API website.

Here are available modes with their short descriptions. Detailed descriptions and examples are in the **User Guide** below.

1. Generating anchors

Mode option: `generate_anchor`

Convert reference into an anchor without checking the website.

2. Find anchor

Mode option: `find_by_anchor`

Parse API website and collect all ids from specific tags. Then convert reference into an anchor and check whether the converted anchor is present among these ids.

3. Find tag content

Mode option: `find_by_tag_content`

This mode searches not by tag ids but by tag content (`<tag id="id">content</tag>`) Parse API website and collect all tags from the specified list, which have ids and text content. The content to search is constructed from the reference. If the tag is found, return a link to its id.

4. Find method in swagger spec for SwaggerUI

Mode option: `find_for_swagger`

Parse the swagger spec file and find the referenced method. The anchor is then constructed by a template. This mode will work for SwaggerUI websites.

5. Find method in swagger spec for Redoc

Mode option: `find_for_redoc`

Parse the swagger spec file and find the referenced method. The anchor is then constructed by a template. This mode will work for Redoc websites.

APIReferences is a highly customizable preprocessor. You can tune almost anything about reference conversion.

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/status` then the endpoint prefix should be stated `/api/v0`. In references you can use either full resource (`{endpoint_prefix}/{command}`) or just the command. APIReferences will sort it out for you.
- **output** – string, which will replace the reference.
- **tag content** – plain text between the tags, for example `<tag>Tag content</tag>`.
- **anchor** – web-element id with leading number sign, for example `#get-user-config`. Adding the anchor to the end of the web URL will make a browser scroll to the specified web element.
- **mode** – the way APIReferences will determine correct anchors to add to website URLs.

Quick Recipes

Recipe 1: find by tag content

We want reference ‘`GET /user/status`’ to be pointed at this element on our API website:

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

Minimal sufficient foliant.yml:

```

1 preprocessors:
2     apireferences:
3         API:
4             My-API:
5                 mode: find_by_tag_content
6                 url: http://example.com/api  # path to your
7                 API website
8                     content_template: 'Operation {verb} {command}
9                 }'

'GET /user/status' -> GET /user/status

```

Recipe 2: find by tag id

The task is the same as in Recipe 1. We want reference ‘GET /user/status’ to be pointed at this element on our API website:

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

Minimal sufficient foliant.yml:

```

1 preprocessors:
2     apireferences:
3         API:
4             My-API:
5                 mode: find_by_anchor
6                 url: http://example.com/api  # path to your
7                 API website
8                     anchor_template: '{verb} {command}'
9                     anchor_converter: slate

'GET /user/status' -> GET /user/status

```

Recipe 3: generate tag id

The task is the same as in Recipes 1 and 2, but this time you don’t have access to API website at the time of building foliant project. We want reference ‘GET /user/status’ to be pointed at this element on our API website:

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

Minimal sufficient foliant.yml:

```
1 preprocessors:
2     apireferences:
3         API:
4             My-API:
5                 mode: generate_anchor
6                 url: http://example.com/api  # path to your
7                 API website
8                 anchor_template: '{verb} {command}'
9                 anchor_converter: slate
```

‘GET /user/status’ -> [GET /user/status](#)

Recipe 4: find link for SwaggerUI

We have a SwaggerUI website and we need to find link to the method by reference ‘GET /user/status’.

Method anchors on SwaggerUI consist of tag and operationId, both of which are not present in our reference. APIReferences can find them for you in the spec file. Let’s assume that correct tag and operationId are usertag and getStatus.

Minimal sufficient foliant.yml:

```
1 preprocessors:
2     apireferences:
3         API:
4             My-API:
5                 mode: generate_for_swagger
6                 url: http://example.com/swagger_ui  # path
7                 to your API website
8                 spec: !path swagger.json  # path or direct
9                 url to OpenAPI spec
```

‘GET /user/status’ -> [GET /user/status](#)

Installation

```
$ pip install foliantcontrib.apireferences
```

Config

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

```
1 preprocessors:  
2   - apireferences
```

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 - apireferences:  
3   targets: # optional. default: []  
4     - site  
5   trim_if_targets: # optional. default: []  
6     - pdf  
7   prefix_to_ignore: Ignore # optional  
8   warning_level: 2 # optional  
9   reference: # optional  
10     - regex: *ref_pattern  
11       only_with_prefixes: false  
12       only_defined_prefixes: false  
13       output_template: '[{verb} {command}]({url})'  
14       trim_template: `'{verb} {command}'  
15     - regex: *another_ref_pattern # second reference  
config. Unlisted options are default  
16       output_template: '**{verb} {command}**'  
17 API: # below are examples for each mode  
18   Client-API: # reference prefix  
19     mode: generate_anchor  
20     url: http://example.com/api/client  
21     anchor_template: '{verb} {command}'  
22     anchor_converter: pandoc # optional  
23     endpoint_prefix: /api/v1 # optional  
24   Admin-API:  
25     mode: find_by_anchor  
26     url: http://example.com/api/admin
```

```

27         anchor_template: '{verb} {command}'
28         anchor_converter: pandoc # optional
29         endpoint_prefix: /api/v1 # optional
30         tags: ['h1', 'h2', 'h3', 'h4'] # optional
31         login: login # optional
32         password: password # optional
33     External-API:
34         mode: find_by_tag_content
35         url: http://example.com/api/external
36         content_template: '{verb} {command}'
37         endpoint_prefix: /api/v1 # optional
38         tags: ['h1', 'h2', 'h3', 'h4'] # optional
39         login: login # optional
40         password: password # optional
41     Internal-API:
42         mode: find_for_swagger
43         url: http://example.com/api/swagger-ui
44         anchor_template: '/{tag}/{operation_id}'
45         anchor_converter: no-transform
46         endpoint_prefix: /api/v1 # optional
47         login: login # optional
48         password: password # optional

```

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: []

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.

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

warning_level (optional) 2 – show all warnings for not found references; 1 – show only warnings for not found prefixed references; 0 – don't show warnings about not found references. Default: 2

reference (optional) List of dictionaries. 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.

All reference properties have defaults. If any of them are missing in the config, the defaults will be used. If **reference** section is omitted, APIReferences will use default values.

Reference options

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

```
'((?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. References without prefixes 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 APIReferences cuts out prefixes. The reference will be replaced with text based on this template. Default: `'`{verb} {command}`'`

API (required) A subsection for listing APIs and their properties. Define a separate subsection for each API here. 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 the preprocessor to work.

API properties

The list of options and some default values differ for each mode.

mode (required) API mode, which determines how references are collected. Available modes: `generate_anchor`, `find_by_anchor`, `find_by_tag_content`, `find_for_swagger`, `find_for_redoc`.

generate_anchor mode

url (required) An API documentation web-page URL. It will be used to construct the full link to the method.

anchor_template (required) A template string describing the format of the anchors in the API documentation web-page. You can use placeholders in {curly braces}, with names of the groups from the reference regex. Example: `'user-content {verb} {command}'`.

anchor_converter (optional) anchor converter from [this list](#). Determines how string `GET /user/status` is converted into `get-userstatus` or `get-user-status` etc. [List of available converters](#). Default: `pandoc`

endpoint_prefix (optional) The endpoint prefix from the server root to API methods. If is stated – APIReferences can divide the command in the reference and search for it more accurately. Also, you could use it in templates. More info in the **Commands and Endpoint Prefixes** section. Default: ''

find_by_anchor mode

url (required) An API documentation web-page URL. It will be used to construct the full link to the method. In this mode, it is also being parsed to check whether the generated anchor is present on the page.

anchor_template (required) A template string describing the format of the anchors in the API documentation web-page. You can use placeholders in {curly braces}, with names of the groups in the reference regex. Example: `'user-content {verb} {command}'`.

anchor_converter (optional) anchor converter from [this list](#). Determines how string GET /user/status is converted into get-userstatus or get-user-status etc. Default: pandoc

endpoint_prefix (optional) The endpoint prefix from the server root to API methods. If is stated – APIReferences can divide the command in the reference and search for it more accurately. Also, you could use it in templates. More info in the **Commands and Endpoint Prefixes** section. Default: ''

tags (optional) list of HTML tags which will be parsed out from the page and searched for ids. Default: ['h1', 'h2', 'h3', 'h4']

login (optional) Login for basic authentication if present on your API site.

password (optional) Password for basic authentication if present on your API site.

find_by_tag_content mode

url (required) An API documentation web-page URL. It will be used to construct the full link to the method. In this mode, it is also being parsed to check whether the generated anchor is present on the page.

content_template (required) A template string describing the format of the tag content in the API documentation web-page. You can use placeholders in {curly braces}, with names of the groups in the reference regex. Example: '{verb} {command}'.

endpoint_prefix (optional) The endpoint prefix from the server root to API methods. If is stated – APIReferences can divide the command in the reference and search for it more accurately. Also you could use it in templates. More info in the **Commands and Endpoint Prefixes** section. Default: ''

tags (optional) list of HTML tags which will be parsed out from the page and searched for ids. Default: ['h1', 'h2', 'h3', 'h4']

login (optional) Login for basic authentication if present on your API site.

password (optional) Password for basic authentication if present on your API site.

find_for_swagger mode

url (required) An API documentation web-page URL. It will be used to construct the full link to the method.

spec (required) URL or local path to OpenAPI specification file.

anchor_template (optional) A template string describing the format of the anchors in the API documentation web-page. You can use placeholders in {curly

braces}, with names of the groups in the reference regex. In this mode, you can also use two additional placeholders: {tag} and {operation_id}. Default: '/{tag}/{operation_id}'.

endpoint_prefix (optional) The endpoint prefix from the server root to API methods. You may use it in output template. Default: ''

login (optional) Login for basic authentication if present on your API site.

password (optional) Password for basic authentication if present on your API site.

find_for_redoc mode

url (required) An API documentation web-page URL. It will be used to construct the full link to the method.

spec (required) URL or local path to OpenAPI specification file.

anchor_template (optional) A template string describing the format of the anchors in the API documentation web-page. You can use placeholders in {curly braces}, with names of the groups in the reference regex. In this mode, you can also use two additional placeholders: {tag} and {operation_id}. Default: 'operation/{operation_id}'.

endpoint_prefix (optional) The endpoint prefix from the server root to API methods. You may use it in output template. Default: ''

login (optional) Login for basic authentication if present on your API site.

password (optional) Password for basic authentication if present on your API site.

User guide

The purpose of APIReferences is to convert references into Markdown links.

Reference is a chunk of text in your Markdown source which will be parsed by APIReferences, separated into groups, and converted into a link. An example of a reference is 'GET /user/authenticate'. APIReferences uses Regular Expressions to find the reference and split into groups. You can supply your own regular expression in **reference -> regex** param (details in **Capturing References** section below). If you are using the default one, the reference from the example above will be split into two groups:

- **verb**: GET,
- **command**: /user/authenticate.

These groups then will be used to find the referenced method on the API website and also to construct an output string.

For example, with `find_by_tag_content` mode (see the detailed description of all modes below) APIReferences will use `content_template` from API configuration to construct a tag content and search for it on the API website. If the content template is '`{verb} {command}`', then the constructed content for the example above will be `GET /user/authenticate`. APIReferences will search for a tag with such content on the page and get its id.

The found tag may be `<h2 id="get-userauthenticate">GET /user/authenticate</h2>`. APIReferences will take the id from this tag and use it as an anchor to the link: `#get-userauthenticate`. Then it will add the API website path and here's your url: `http://example.com/api/#get-userauthenticate`.

Now, when APILink has the url of the method description, it can construct an output string. The output string is formed by a template, stated in reference `output_template` param. This template contains placeholders, which correspond to the reference groups with an addition of `{url}` placeholder, which contains the url formed above.

If the output template is '`[{verb} {command}]({url})`', then the output string for our example will be:

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

That's it, we've turned our reference into a Markdown link:

```
' GET /user/authenticate' -> [GET /user/authenticate](http://example.com/api/#get-userauthenticate).
```

That's the big picture. Now let's start with exploring different modes by means of which APIReferences captures references on API websites and transforms them into links.

API Modes

As mentioned above, APIReferences takes a reference from your markdown source and splits it into groups. It then uses these groups to find the correct id on the API website. How this search is performed is determined by API Mode. It can search for a specific tag on the page by tag content or by its id; it can also search for the operation in an OpenAPI specification file or just construct an id without any checks, depending on the mode you've chosen. The mode is specified in `API -> <api name> -> mode config` option.

generate_anchor mode

`generate_anchor` is the simplest mode. It just generates the anchor basing on the `anchor_template` parameter. It doesn't perform any checks on the API website and doesn't even require the website to be reachable at the time of building your Foliant project.

Let's assume that your API website code looks like this:

```
1 ...
2 <h2 id="user-content-get-userlogin">GET /user/status</h2>
3 <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit.
</p>
4
5 <h2 id="user-content-get-apiv2adminstatus">GET /api/v2/admin
 /status</h2>
6 <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit.
</p>
7 ...
```

APIReferences config in your `foliant.yml` in this case may look like this:

```
1 preprocessors:
2     apireferences:
3         API:
4             My-API:
5                 mode: generate_anchor
6                 url: http://example.com/api
7                 anchor_template: 'user content {verb} {
command}'
8                 anchor_converter: pandoc
```

As you may have noticed, there's no `reference` section in the example above. That's because we will be using default values for the reference.

Now let's reference a `GET /user/status` method in our Markdown source:

```
To find out user's status use `My-API: GET /user/status`  
method.
```

Note that for `generate_anchor` mode, the API prefix (My-API in our case) is required in the reference. More info about prefixes in [Handling Multiple APIs](#) section.

APIReferences will notice a reference mentioned in our markdown: ' My-API: GET /user/status'. It will capture it and split into three groups:

- **prefix**: My-API,
- **verb**: GET,
- **command**: /user/status.

Then it will pass it to the anchor template '`user content {verb} {command}`' which we've stated in our config, and this will result in a string:

```
'user content GET /user/status'
```

After that APIReferences will convert this string into an id with anchor converter. We've chosen pandoc converter in our config, which will turn the string into this: `user-content-getuserstatus`. That's exactly the id we needed, look at the webpage source:

```
<h2 id="user-content-get-userstatus">GET /user/status</h2>
```

APIReferences will add this id to our API url (which we've stated in config) to form a link: `http://example.com/api#user-content-get-userstatus`.

Finally, it's time to construct a Markdown link. APIReferences takes an `output_template` from the reference config (which is omitted in our example foliant.yml because we are using defaults): '`[{verb} {command}]({url})`'.

Placeholders in the output template are replaced by groups from our reference, except `{url}` placeholder which is replaced with the url constructed above:

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

The conversion is done. Our Markdown content will now look like this:

```
To find out user's status use [GET /user/status](http://example.com/api#user-content-get-userstatus) method.
```

find_by_anchor mode

`find_by_anchor` generates the id by `anchor_template` parameter and searches for this id on the API web page. If an element with such id is found, the reference is converted into a Markdown link. If not – the reference is skipped.

Let's assume that your API website code looks like this:

```
1 ...
2 <h2 id="api-method-get-userstatus">GET /user/status</h2>
3 <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit.
</p>
4
5 <h2 id="api-method-get-apiv2adminstatus">GET /api/v2/admin/
status</h2>
6 <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit.
</p>
7 ...
```

APIReferences config in your `foliant.yml` in this case may look like this:

```
1 preprocessors:
2     apireferences:
3         reference:
4             output_template: '**[{verb} {command}]({url})**'
5             # other reference properties are default
6 API:
7     My-API:
8         mode: find_by_anchor
9         url: http://example.com/api
10        tags: ['h1', 'h2']
11        anchor_template: 'api-method {verb} {command}'
12    }
13
14    anchor converter: pandoc
```

Now let's reference a **GET /user/status** method in our Markdown source:

To find out user's status use 'GET /user/status' method.

APIReferences will notice a reference mentioned in our markdown: ‘ GET /user/status’. It will capture it and split into two groups:

- **verb**: GET,
- **command**: /user/status.

Then it will pass it to the anchor template '`api-method {verb} {command}`' which we've stated in our config, and this will result in a string:

```
'user content GET /user/status'
```

After that APIReferences will convert this string into an id with an anchor converter. We've used pandoc converter in our config, which will turn the string into this: `api-method-getuserstatus`.

Now APIReferences will parse the web page and look for all `h1` and `h2` tags (as specified in `tags` parameter) that have ids and compare these ids to our generated id.

One of the elements satisfies the requirement:

```
<h2 id="api-method-get-userstatus">GET /user/status</h2>
```

It means that referenced method is present on API web page, so APIReferences will add this id to our API url (which we've stated in config) to form a link: `http://example.com/api#api-method-get-userstatus`.

Finally, it's time to construct a Markdown link. APIReferences takes an `output_template` from the reference config: `'**[{verb} {command}]({url})**'`.

Placeholders in the output template are replaced by groups from our reference, except `{url}` placeholder which is replaced with the url constructed above:

```
**[GET /user/status](http://example.com/api#api-method-get-userstatus)**
```

The conversion is done. Our Markdown content will now look like this:

```
To find out user's status use **[GET /user/status](http://example.com/api#api-method-get-userstatus)** method.
```

`find_by_tag_content` mode

`find_by_tag_content` generates tag content by the `content_template` and searches for an HTML element with such content on the API web page. If an element is found, the reference is converted into a Markdown link. If not – the reference is skipped.

This mode is convenient when there's no way to determine tag id basing on the reference, for example, when ids are random strings.

Let's assume that your API website code looks like this:

```
1 ...
2 <h2 id="o1egwb7agw">GET /user/status</h2>
3 <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit.
</p>
4
5 <h2 id="y3yn8ewg32">GET /api/v2/admin/status</h2>
6 <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit.
</p>
7 ...
```

APIReferences config in your `foliant.yml` in this case may look like this:

```
1 preprocessors:
2     apireferences:
3         reference:
4             output_template: '[{prefix}: {verb} {command}]({url})'
5                 # other reference properties are default
6         API:
7             My-API:
8                 mode: find_by_tag_content
9                 url: http://example.com/api
10                tags: ['h1', 'h2']
11                content_template: '{verb} {command}'
```

Now let's reference a **GET /user/status** method in our Markdown source:

```
To find out user's status use `My-API: GET /user/status`  
method.
```

APIReferences will notice a reference mentioned in our markdown: 'My-API: GET /user/status'. The reference has the prefix My-API, which means that My-API from the API section should be used. It will capture it and split into three groups:

- **prefix**: My-API,
- **verb**: GET,

– **command:** /user/status.

Then it will pass it to the header template '{verb} {command}' which we've stated in our config, and this will result in a string:

```
'GET /user/status'
```

Now APIReferences will parse the web page and look for all `h1` and `h2` tags (as specified in the `tags` parameter) whose content equals to our generated content.

One of the elements satisfies the requirement:

```
<h2 id="o1egwb7agw">GET /user/status</h2>
```

It means that referenced method is present on the API web page, so APIReferences will take an id `o1egwb7agw` from it and add it to our API url (which we've stated in config) to form a link: `http://example.com/api#o1egwb7agw`.

Finally, it's time to construct a Markdown link. APIReferences takes an `output_template` from the reference config: '[{prefix}: {verb} {command}]({url})'.

Placeholders in the output template are replaced by groups from our reference, except `{url}` placeholder which is replaced with the url constructed above:

```
[My-API: GET /user/status](http://example.com/api#api-method-get-userstatus)
```

The conversion is done. Our Markdown content will now look like this:

```
To find out user's status use [My-API: GET /user/status](  
http://example.com/api#api-method-get-userstatus) method.
```

find_for_swagger mode

`find_for_swagger` mode parses the OpenAPI spec file and looks for the referenced method in it. It then generates an anchor for SwaggerUI website based on data from the reference and the operation properties in the spec.

Let's assume that your OpenAPI specification looks like this:

```
1 {
2     "swagger": "2.0",
3     ...
4     "paths": {
```

```

5     "/user/status": {
6         "GET": {
7             "tags": ["userauth"],
8             "summary": "Returns user auth status",
9             "operationId": "checkStatus",
10            ...
11        },
12    }
13    ...

```

On the default SwaggerUI website the anchor to this method will be `#/userauth/checkStatus`. It consists of the first tag from the operation properties and the operationId. So to generate the proper anchor APIReferences will need to get those parts from the spec.

APIReferences config in your `foliant.yml` in this case may look like this:

```

1 preprocessors:
2     apireferences:
3         # reference options are default in this example
4         API:
5             My-API:
6                 mode: find_for_swagger
7                 url: http://example.com/api
8                 spec: !path swagger.json
9                 anchor_template: '/{tag}/{operation_id}' # you can omit this line because it's the default value

```

Now let's reference a **GET /user/status** method in our Markdown source:

```
To find out user's status use `GET /user/login` method.
```

APIReferences will notice a reference mentioned in our markdown: ' GET /user/login'. It will capture it and split into two groups:

- **verb**: GET,
- **command**: /user/login.

Note that `verb` and `command` groups are required for this mode if you are to redefine default reference regex.

Now, when we have a verb and a command, we can search for it in the OpenAPI spec. APIReferences parses the spec and searches the `paths` section for our operation. From the operation properties APIReferences takes two values:

- **tag**: first element from the `tags` list,
- **operationId**.

These values are then passed to the anchor template `'/{tag}/{operation_id}'`, along with groups from our reference, this will result in a string:

```
'/userauth/checkStatus'
```

That's the id we were looking for. APIReferences will add it to our API url (which we've stated in config) to form a link: `http://example.com/api#/userauth/checkStatus`.

Finally, it's time to construct a Markdown link. APIReferences takes an `output_template` from the reference config, which is default: `'[{{verb}} {{command}}]({{url}})'`.

Placeholders in the output template are replaced by groups from our reference, except `{url}` placeholder which is replaced with the url constructed above:

```
[GET /user/login](http://example.com/api#/userauth/checkStatus)
```

The conversion is done. Our Markdown content will now look like this:

```
To find out user's status use [GET /user/login](http://example.com/api#/userauth/checkStatus) method.
```

`find_for_redoc` mode

`find_for_redoc` is similar to `find_for_swagger` mode, except that deafult anchor template is `'operation/{operation_id}'`.

Handling Multiple APIs

APIReferences can work with several APIs at once, and honestly, it's very good at this.

Let's consider an example `foliant.yml`:

```
1 preprocessors:  
2     apireferences:  
3         API:
```

```

4      Client-API:
5          mode: find_by_tag_content
6          url: http://example.com/api/client
7          content_template: '{verb} {command}'
8      Admin-API:
9          mode: find_by_anchor
10         url: http://example.com/api/admin
11         content_template: '{verb} {command}'

```

In this example we've defined two APIs: Client-API and Admin-API, these are just names, they may be anything you want. Now we can reference both APIs:

- `1 When user clicks "LOGIN" button, the app sends a request `POST /user/login`.`
- `2`
- `3 To restrict user from logging in run `PUT /admin/ban_user/{id}`.`

After applying the preprocessor, this source will turn into:

- `1 When user clicks "LOGIN" button, the app sends a request [POST /user/login](http://example.com/api/client#post-userlogin).`
- `2`
- `3 To restrict user from logging in run [PUT /admin/ban_user/{id}]()](http://example.com/api/admin#put-adminbanuser-id).`

As you see, APIReferences determined, which reference corresponds to which API. That is possible because when APIReferences meets a non-prefixed reference, it goes through each defined API and searches for the mentioned method.

But what happens if we reference a method which is present in both APIs?

Run `GET /system/healthcheck` for debug information.

You have to understand that, even though APIReferences is very powerful, it doesn't understand the concept of free will. It can't make the choice for you, so instead, it will show a warning and skip this reference:

```
WARNING: [index.md] Failed to process reference. Skipping. `GET /system/healthcheck` is present in several APIs (Client-API, Admin-API). Please, use prefix.
```

In the warning text, there's a suggestion to use a prefix. A prefix is a way to make your reference more specific and point APIReferences to the correct API. The value of the prefix is the API name as defined in the config. So for Client API, the prefix would be **Client-API**, for Admin – **Admin-API**. Let's fix our example:

- 1 Run `Admin-API: GET /system/healthcheck` to get debug information about the Admin API service.
- 2
- 3 Run `Client-API: GET /system/healthcheck` to get debug information about the Client API service.

If you don't like the format in which we supply prefix (`<prefix>: <verb> <command>`), you can change it by tweaking reference regex. More info in **Capturing References** section.

It's recommended to always use prefixes for unambiguity. The `generate_anchor` mode won't work at all for references without prefixes, because it doesn't perform any checks and almost always returns a link.

Handling Multiple Reference Configuration

You can not only make APIReferences work with different APIs but also with different reference configurations. `reference` parameter is a list for a reason. And because `output_template` is part of reference configuration, you can make different references transform into different values.

Here's an example config:

```
1 preprocessors:  
2     apireferences:  
3         reference:  
4             - only_with_prefixes: true  
5                 output_template: '**[{verb} {command}]({url})'  
6             - only_with_prefixes: false  
7                 output_template: '[{verb} {command}]({url})'
```

```
8     API:  
9     ...
```

With such config references with prefixes will be transformed into **bold links**, while non-prefixed references will remain regular links.

Commands and Endpoint Prefixes

APIReferences treats the `command` part of your reference in a special way. While searching for it on the API website it will try to substitute the command place holder:

- with and without leading slash (`/user/login` and `user/login`),
- with and without endpoint prefix, if one is defined (`/api/v1/user/login` and `/user/login`).

Here's an example config to illustrate this feature:

```
1 preprocessors:  
2     apireferences:  
3         reference:  
4             - only_with_prefixes: true  
5                 output_template: '**[{verb} {command}]({url})'  
6             **!  
7                 - only_with_prefixes: false  
8                     output_template: '[{verb} {command}]({url})'  
9         API:  
10            My-API:  
11                mode: find_by_tag_content  
12                url: http://example.com/api  
13                content_template: '{verb} {command}'  
14                endpoint_prefix: /api/v1
```

Considering that the API website source looks like this:

```
<h2 id="asoi17uo">GET /api/v1/user/status</h2>
```

Which of these references, do you think, will give us the desired result?

- 1 `GET /user/status`
- 2 `GET user/status`
- 3 `GET /api/v2/user/status`

If you were reading carefully, you already know the answer – all of these references will result in the same link:

```
1 [GET /user/status](http://example.com/api#asoil7uo)
2 [GET /user/status](http://example.com/api#asoil7uo)
3 [GET /user/status](http://example.com/api#asoil7uo)
```

Capturing References

APIReferences uses regular expressions to capture references to API methods in Mark-down files.

The default reg-ex is:

```
`((?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 the default expression uses [Named Capturing Groups](#). You have to use them too if you are to redefine the expression. You can name these groups as you like and have as many or as few as you wish, but it's recommended to include the `prefix` group for API prefix logic to work. It is also required for all groups which are in the `output_template` also to be present in the regex.

To redefine the regular expression add an option `regex` to the reference config.

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

```
1 preprocessors:
2   - apireferences:
3     reference:
4       - regex: `((?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 reference configuration.

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

For example, look at the default template:

```
1 preprocessors:  
2   - apireferences:  
3     reference:  
4       - output_template: '[{verb} {command}]({url})',
```

Don't forget the single quotes around the template. These braces and parenthesis easily make YAML think that it is an embedded dictionary or list.

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 want references to be transformed into something else, create your own template. You can use placeholders from the reference regular expression along with some additional:

placeholder	description	example
source	Full original reference string	'Client-API: GET user/info'
url	Full url to the method description	http://example. com/api/#get- user-info
endpoint_prefix	API endpoint prefix from API configuration	/api/v2

Placeholders from the default regex are:

placeholder	description	example
prefix	API Prefix used in the reference	Client-API
verb	HTTP verb used in the reference	GET
command	API command being referenced with endpoint prefix removed	/user/info

Archeme

pypi v1.0.3

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     config_concat: false
9     config_file: null
10    action: generate
11    format: png
12    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`.

config_concat Flag that tells the preprocessor to read the config file and the diagram definition as YAML strings, concatenate these strings, and then load the concatenation result, i.e. single YAML string, as an object. If this option is not set (by default), config and diagram definition will be loaded as separate objects, and then merged. This option may be useful when some aliases are defined in the config, and you would like to use their values in the diagram definition.

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.

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](#)).

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`. The value of this option may be overridden by the respective `format` tag attribute.

targets Allowed targets for the preprocessor. If not specified (by default), the preprocessor 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:

- `caption`—to set an alternative text description of the diagram that may be rendered as image caption by some backends;
- `module_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 module_id="one" caption="Module 1" 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 module_id="...">` tags, you should omit the `file` and `description` module parameters in your combined diagrams definitions.

Argdown

[pypi v0.1.1](#)

[github v0.1.1](#)

Argdown Diagrams Preprocessor for Foliant

[Argdown](#) is a modeling language for creating argument maps. This preprocessor takes Argdown diagram definitions in source markdown files and converts them into images on the fly during project build.

This preprocessor uses [Argdown Image Export package](#) tool by [Christian Voigt](#) to convert diagrams into images.

Installation

```
$ pip install foliantcontrib.argdown
```

You will also need to install Argdown CLI and the Image Export package:

```
1 $ npm install -g @argdown/cli  
2 $ npm install -g @argdown/image-export
```

Config

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

```
1 preprocessors:  
2     - argdown
```

The preprocessor has a number of options:

```
1 preprocessors:  
2     - argdown:  
3         cache_dir: !path .diagramscache/argdown  
4         converter_path: argdown  
5         format: png  
6         as_image: true  
7         params:  
8             no-title: true  
9             fix_svg_size: false
```

cache_dir Path to the cache directory for the generated diagrams. It can be a path relative to the project root or a global one.

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

converter_path Path to Argdown CLI. By default, it is assumed that you have the `argdown` command in your PATH, but if it is not the case you can define it here.
Default: `argdown`

format Output format of the diagram image. Available formats at the time of writing:
`dot`, `graphml`, `svg`, `pdf`, `png`, `jpg`, `webp`. Default: `png`

as_image If `true` – inserts the diagram into the document as Markdown-image.
If `false` – inserts the source code of the diagram directly into the document
(works only for `svg`, `dot` and `graphml` formats). Default: `true`

params Params passed to the Argdown CLI map tool. Value of this option must be a YAML-mapping. Params which require values should be specified as `param: value`; params which don't require values should be specified as `param: true`.

To see the full list of available params, run `argdown map --help`.

fix_svg_size Works only with `svg` format and `as_image: false`. By default `svg` is embedded with hardcoded width and height so they may exceed the boundaries of your HTML page. If this option is set to `true` the `svg` width and height will be set to `100%` which will make it fit inside your content container. Default: `false`.

Usage

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

```
1 ,
2 Heres the diagram:
3
4 <argdown>
5 ===
6 title: The Core Argument of Populism
7 author: David Lanius
8 date: 27/10/2018
9 ===
```

```
10
11
12 This is a reconstruction of right-wing populist argumentation
13 based on the electoral platform of the German party...
14 </argdown>
```

You can override preprocessor parameters in the tag options. For example if the format for diagrams is set to `png` in `foliant.yml` and you need one of your diagrams to render in `svg`, override the `format` option in the tag:

```
1 SVG diagram:
2
3 <argdown format="svg">
4 ...
5 </argdown>
```

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

```
1 Diagram with a caption:
2
3 <argdown caption="Diagram of the opposing arguments">
4 ...
5 </argdown>
```

Badges

pypi v1.0.3

GitHub v1.0.3

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 `preprocessors` 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_delimiter: '\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_delimiter 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 `preprocessors` 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>
```

BPMN

[pypi v1.0.1](#)

[github v1.0.1](#)

BPMN Diagrams Preprocessor for Foliant

[BPMN \(Business Process Modeling Notation\)](#) is visual modeling language for documenting business workflows. This preprocessor converts BPMN diagram definitions in source markdown files and converts them into images on the fly during project build.

This preprocessor uses [bpmn-to-image](#) tool by [bpmn.io](#) to convert diagrams into images.

Installation

```
$ pip install foliantcontrib.bpmn
```

You will also need to install bpmn-to-image:

```
$ npm install -g bpmn-to-image
```

Config

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

```
1 preprocessors:  
2   - bpmn
```

The preprocessor has a number of options:

```
1 preprocessors:  
2   - bpmn:  
3     cache_dir: !path .diagramscache/bpmn  
4     converter_path: bpmn-to-image  
5     format: png  
6     as_image: true  
7     params:  
8       no-title: true  
9     `fix_svg_size`: false
```

cache_dir Path to the cache directory for the generated diagrams. It can be a path relative to the project root or a global one.

To save time during build, only new and modified diagrams are rendered.

The generated images are cached and reused in future builds.

converter_path Path to bpmn-to-image binary. By default, it is assumed that you have the `bpmn-to-image` command in your `PATH`, but if it is not the case you can define it here. Default: `bpmn-to-image`

format Output format of the diagram image. [Available formats](#) at the time of writing: `pdf`, `png`, `svg` (note that most backends won't render `pdf` as image). Default: `png`

as_image If `true` – inserts the diagram into the document as Markdown-image. If `false` – inserts the `svg` code of the diagram directly into the document (works only for `svg` format). Default: `true`

params Params passed to the bpmn-to-image tool. Value of this option must be a YAML-mapping. Params which require values should be specified as `param: value`; params which don't require values should be specified as `param: true`:

```
1 preprocessors:
2   - bpmn:
3     params:
4       no-footer: true
5       min-dimensions: '500x300'
```

To see the full list of available params, run the `bpmn-to-image` command without parameters.

fix_svg_size Works only with `svg` format and `as_image: false`. By default `svg` is embedded with hardcoded width and height so they may exceed the boundaries of your HTML page. If this option is set to `true` the `svg` width and height will be set to `100%` which will make it fit inside your content container. Default: `false`.

Usage

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

```
1 '
2 Heres the diagram:
3
4 <bpmn>
5   <?xml version="1.0" encoding="UTF-8"?>
6   <definitions xmlns="http://www.omg.org/spec/BPMN
/20100524/MODEL" xmlns:bpmndi="http://www.omg.org/spec/BPMN
/20100524/DI" xmlns:omgdc="http://www.omg.org/spec/DD
/20100524/DC" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" id="sid-38422fae-e03e-43a3-bef4-bd33b32041b2"
targetNamespace="http://bpmn.io/bpmn" exporter="http://bpmn.
io" exporterVersion="0.10.1">
7     <process id="Process_1" isExecutable="false">
8         <task id="Task_0l0q2kz" name="Single Task" />
```

```

9      </process>
10     <bpmndi:BPMDiagram id="BpmnDiagram_1">
11         <bpmndi:BPMNPlane id="BpmnPlane_1" bpmnElement="Process_1">
12             <bpmndi:BPMSShape id="Task_0l0q2kz_di" bpmnElement="Task_0l0q2kz">
13                 <omgdc:Bounds x="206" y="108" width="100" height="80" />
14             </bpmndi:BPMSShape>
15         </bpmndi:BPMNPlane>
16     </bpmndi:BPMDiagram>
17 </definitions>
18 </bpmmn>
```

You can override preprocessor parameters in the tag options. For example if the format for diagrams is set to `png` in `foliant.yml` and you need one of your diagrams to render in `svg`, override the `format` option in the tag:

```

1 SVG diagram:
2
3 <bpmmn format="svg">
4 ...
5 </bpmmn>
```

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

```

1 Diagram with a caption:
2
3 <bpmmn caption="Diagram of the supply process">
4 ...
5 </bpmmn>
```

Confluence

[pypi](#) v0.6.20

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: !CONFLUENCE_PASS
7     space_key: "~user"
8     pandoc_path: pandoc
9     verify_ssl: true
```

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.

It is not secure to store plain text passwords in your config files. We recommend to use [environment variables](#) to supply passwords

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).

verify_ssl If `false`, SSL verification will be turned off. Sometimes when dealing with Confluence servers in Intranets it's easier to turn this option off rather than fight with admins. Not recommended to turn off for public servers in production.
Default: `true`

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

[pypi](#) v1.0.2

[GitHub](#) v1.0.2

CSVTables for Foliant

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

[pypi](#) v1.0.7

[GitHub](#) v1.0.7

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](#) v0.3.1

[GitHub](#) v0.3.1

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](#) v0.1.8

[GitHub](#) v0.1.8

Database Documentation Generator for Foliant

The screenshot shows a database documentation interface. On the left, a sidebar lists tables: COUNTRIES, DEPARTMENTS, EMPLOYEES, JOBS, JOB_HISTORY, LOCATIONS, and REGIONS. The 'REGIONS' table is selected and highlighted in blue. The main content area has a header 'REGIONS'. Below it is a detailed description: 'Regions table that contains region numbers and names. Contains 4 rows; references with the Countries table.' A table defines the columns:

column	nullable	type	descr	fkey
REGION_ID	N	NUMBER	Primary key of regions table.	
REGION_NAME	Y	VARCHAR2	Names of regions. Locations are in the countries of these regions.	

Below this is a 'Views' section. It shows a view named 'EMP_DETAILS_VIEW' with the following details:

Name: EMP_DETAILS_VIEW
Schema: HR

```
SELECT
    e.employee_id,
    e.job_id,
    e.manager_id,
    e.department_id,
    d.location_id,
    l.country_id,
    e.first_name,
    e.last_name,
    e.salary,
    e.commission_pct,
```

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](#),
- [Microsoft SQL Server](#),
- [MySQL](#).

Important Notice: We, here at Foliant, don't work with all of the databases mentioned above. That's why we cannot thoroughly test the preprocessor's work with all of them. That's where we need your help: If you encounter any errors during build; if you are not getting enough information for your document in the template; if you can't make the filters work; or if you see any other anomaly, please [send us an issue](#) in GitHub. We will try to fix it as fast as we can. Thanks!

Installation

Prerequisites

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

PostgreSQL

To install PostgreSQL simply run

```
$ pip3 install psycopg2-binary
```

Oracle

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.

Next install the Python connector for Oracle database

```
$ pip3 install cx_Oracle
```

Microsoft SQL Server

On Windows you will need to install MS SQL Server.

On Unix you will first need to install [unixODBC](#), and then – the ODBC driver. Microsoft has a detailed instructions on how to install the driver [on Linux](#) and [on Mac](#).

Install the Python connector for Microsoft SQL Server database

```
$ pip3 install pyodbc
```

MySQL

On Mac you can simply run

```
$ brew install mysql
```

On Linux you will have to install server and client packages, for example, with apt-get

```
1 sudo apt-get update
```

```
2 sudo apt-get install -y mysql-server libmysqlclient-dev
```

Finally, install the Python connector for Microsoft SQL Server database

```
$ pip3 install mysqlclient
```

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         dbms: postgres
4         host: localhost
5         port: 5432
6         dbname: postgres
7         user: postgres
8         password: !env DBDOC_PASS
9         doc: True
10        scheme: True
11        filters:
12            ...
13        doc_template: dbdoc.j2
14        scheme_template: scheme.j2
15        components:
16            - tables
17            - functions
18            - triggers
19        driver: '{ODBC Driver 17 for SQL Server}'
```

dbms Name of the DBMS. Should be one of: `pgsql`, `oracle`, `sqlserver`, `mysql`. Only needed if you are using `<dbdoc>` tag. If you are using explicit tags (`<oracle>`, `<pgsql>`), this parameter is ignored.

host Database host address. Default: `localhost`

port Database port. Default: 5432 for `pgsql`, 1521 for Oracle, 1433 for MS SQL, 3306 for MySQL.

dbname Database name. Default: `postgres` for `pgsql`, `orcl` for oracle, `mssql` for MS SQL, `mysql` for MySQL.

user Database user name. Default: `postgres` for `pgsql`, `hr` for oracle, `SA` for MS SQL, `root` for MySQL.

password Database user password. Default: `postgres` for `pgsql`, `oracle` for oracle, `<YourStrong@Passw0rd>` for MS SQL, `passwd` for MySQL.

It is not secure to store plain text passwords in your config files. We recommend to use [environment variables](#) to supply passwords

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'`.

driver Specific option for MS SQL Server database. Defines the driver connection string. Default: `{ODBC Driver 17 for SQL Server}`.

Usage

DBDoc currently supports four database engines: Oracle, PostgreSQL, MySQL and Microsoft SQL Server. 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>
```

To generate MySQL database documentation, add a `<mysql></mysql>` tag to a desired place of your chapter.

```
1 # Introduction  
2  
3 This document contains the most awesome automatically  
generated documentation of our marvellous SQL Server  
database.  
4  
5 <mysql></mysql>
```

To generate SQL Server database documentation, add a `<sqlserver></sqlserver>` tag to a desired place of your chapter.

```
1 # Introduction  
2  
3 This document contains the most awesome automatically  
generated documentation of our marvellous SQL Server  
database.  
4  
5 <sqlserver></sqlserver>
```

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

Note: `regex` and `not_regex` are not supported with Microsoft SQL Server DBMS.

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](#).
- [Default MySQL doc template](#).
- [Default MySQL scheme template](#).
- [Default SQL Server doc template](#).
- [Default SQL Server scheme template](#).

Troubleshooting

If you get errors during build, especially errors concerning connection to the database, you have to make sure that you are supplying the right parameters.

There may be a lot of possible causes for errors. For example, MS SQL Server may fail to connect to local database if you specify host as `localhost`, you have to explicitly write `0.0.0.0` or `127.0.0.1`.

So your first action to root the source of your errors should be running a python console and trying to connect to your database manually.

Here are sample snippets on how to connect to different databases.

PostgreSQL

[psycopg2](#) library is required.

```
1 import psycopg2
2
3 con = psycopg2.connect(
4     "host=localhost "
```

```
5      "port=5432 "
6      "dbname=MyDatabase "
7      "user=postgres"
8      "password=postgres"
9 )
```

Oracle

`cx_Oracle` library is required.

```
1 import cx_Oracle
2
3 con = cx_Oracle.connect(
4     "Scott/Tiger@localhost:1521/MyDatabase",
5     encoding='UTF-8',
6     nencoding='UTF-8'
7 )
```

MySQL

`mysqlclient` library is required.

```
1 from MySQLdb import _mysql
2
3 con = _mysql.connect(
4     host='localhost',
5     port=3306,
6     user='root',
7     passwd='password',
8     db='MyDatabase'
9 )
```

Microsoft SQL Server

`pyodbc` library is required.

```
1 import pyodbc
2
3 con = pyodbc.connect(
4     "DRIVER={ODBC Driver 17 for SQL Server};"
5     "SERVER=0.0.0.0,1433;"
```

```
6     "DATABASE=MyDatabase;"  
7     "UID=Username;PWD=Password_0"  
8 )
```

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 `processors` 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 `&`, `<` with `<`, `>` with `>`, 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_ELASTICSEARCH` 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    }
26 }
```

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 18.** 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](#) v1.0.4

[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 v1.1.5](#)

[github v1.1.5](#)

## 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](#) v1.1.13

[GitHub](#) v1.1.13

# 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>foopath/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. The referenced heading is included.

**to\_heading** Full content of the ending heading when it's necessary to include some part of the referenced file content. The referenced heading will not be included.

**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. The referenced id is included.

**NOTE:** If you want `from_id` and `to_id` features to work with [anchors](#), make sure that anchors preprocessor is listed after includes in foliant.yml.

**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`. The referenced id will not be included.

**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>foopath/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 <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](#) v1.0.2

[GitHub](#) v1.0.2

# 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 v0.1.3](#)

[GitHub v0.1.3](#)

## 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.
Nesciunt, atque.
7
8 ## Subtitle
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: '
'
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 shrunk to this width in symbols. This parameter affects scaling - change it if table columns are merging.

**keep\_narrow\_tables** If true narrow tables will not be stretched to minimum table width.

**table\_columns\_to\_scale** Minimum amount of columns to process the table.

**enable\_hyphenation** Switch breaking text in table cells with the tag set in hyph\_combination. Good for lists, paragraphs, etc.

**hyph\_combination** Custom tag to break a text in multiline tables.

**convert\_to\_grid** If true tables will be converted to the grid format, that allows arbitrary cell' content (multiple paragraphs, code blocks, lists, etc.).

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

## Usage

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

# Pgsqldoc

pypi v1.1.7

GitHub v1.1.7

## 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 `processors` 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

operator	SQL equivalent	description	value
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>/scheme.j2` for database scheme source template.

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

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

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

## Plantuml

pypi v1.0.10

GitHub v1.0.10

## 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 format: png
6 params:
7 ...
8 parse_raw: true
9 as_image: 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.

**format** Diagram format, [list of supported formats](#). Default: `png`.

Another way to specify format is to use `t<format>` option in `params`.

**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`.

**as\_image** If `true` – inserts scheme into document as md-image. If `false` – inserts the file generated by PlantUML directly into the document (only for `svg` format). Default: `true`

## 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 values from the preprocessor config for each diagram.

```
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 v1.0.2

## 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
4 documentation of our API.
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.

## Reindexer

This extension allows to integrate Foliant-managed documentation projects with the in-memory DBMS [Reindexer](#) to use it as a fulltext search engine.

The main part of this extension is a preprocessor that prepares data for a search index. In addition, the preprocessor performs basic manipulations with the database and the namespace in it.

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.reindexer
```

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 `reindexer` to `processors` section in the project config:

```
1 preprocessors:
2 - reindexer
```

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

```
1 preprocessors:
2 - reindexer:
```

```

3 reindexer_url: http://127.0.0.1:9088/
4 insert_max_bytes: 0
5 database: ''
6 namespace: ''
7 namespace_renamed: ''
8 fulltext_config: {}
9 actions:
10 - drop_database
11 - create_database
12 - create_namespace
13 - insert_items
14 use_chapters: true
15 format: plaintext
16 escape_html: true
17 url_transform:
18 - '/?index\.md$': '/'
19 - '\.md$': '/'
20 - '^([^\/]+)': '/\g<1>'
21 require_env: false
22 targets: []

```

**reindexer\_url** URL of your Reindexer instance. “Root” server URL should be used here, do not add any endpoints such as `/api/v1/db` to it.

**insert\_max\_bytes** Reindexer itself or a proxy server may limit the available size of request body. Use this option, if it’s needed to split a large amount of content for indexing into several chunks, so each of them will be sent in a separate request. The value of this option represents maximum size of HTTP POST request body in bytes. Allowed values are positive integers starting from 1024, and 0 (default) meaning no limits.

**database** Name of the database that is used to store your search index.

**namespace** Name of the namespace in the specified database. Namespace in Reindexer means the same as table in relational databases. To store the search index for one documentation project, single namespace is enough.

**namespace\_renamed** New namespace name to be applied if the `rename` option is used; see below.

**fulltext\_config** The value of the `config` field that refers to the description of the composite fulltext index over the `title` and `content` data fields. Used data structure is described below. [Fulltext indexes config options](#) are listed in the Reindexer's official documentation.

**actions** Sequence of actions that the preprocessor should to perform. Available item values are:

- `drop_database`—fully remove the database that is specified as the value of the `database` option. Please be careful using this action when the single database is used to store multiple namespaces. Since this action is included to the default actions list, it's recommended to use separate databases for each search index. The default list of actions assumes that in most cases it's needed to remove and then fully rebuild the index, and wherein the database and the namespace may not exist;
- `create_database`—create the new database with the name specified as the `database` option value;
- `drop_namespace`—delete the namespace that is specified as the `namespace` option value. All `*_namespace` actions are applied to the existing database with the name from the `database` option;
- `truncate_namespace`—remove all items from the namespace that is specified as the `namespace` option value, but keep the namespace itself;
- `rename_namespace`—rename the existing namespace that has the name specified as the `namespace` option value, to the new name from the `renamed_namespace` option. This action may be useful when a common search index is created for multiple Foliant projects, and the index may remain incomplete for a long time during their building;
- `create_namespace`—create the new namespace with the name from the `namespace` option;
- `insert_items`—fill the namespace that is specified in the `namespace` option, with the content that should be indexed. Each data item added to the namespace corresponds a single Markdown file of the documentation project.

**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 Markdown 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 `&gt;`, and `"` with `&quot;`.

**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_REINDEXER` environment variable must be set to allow the preprocessor to perform any operations with the database and the namespace managed by Reindexer. This flag may be useful in CI/CD jobs.

**targets** Allowed targets for the preprocessor. If not specified (by default), the pre-processor applies to all targets.

## Usage

The preprocessor reads each source Markdown file and prepares three fields for indexing:

- `url`—target page URL. This field is used as the primary key, so it must be unique;
- `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 Reindexer API to insert data items (each item corresponds a single Markdown file) into the specified namespace.

Also the preprocessor may call Reindexer API to manipulate the database or namespace, e.g. to delete previously created search index.

You may perform custom search requests to Reindexer API.

The [simple client-side Web application example](#) that is provided as a part of this extension, sends to Reindexer queries like this:

```
1 {
2 "namespace": "testing",
3 "filters": [
4 {
5 "field": "indexed_content",
6 "cond": "EQ",
7 "value": "@title^3 foliant"
8 }
9],
10 "select_functions": [
11 "content = snippet(, , 100, 100, '\n\n')"
12],
13 "limit": 50
14 }
```

To learn how to write efficient queries to Reindexer, you may need to refer to its official documentation on topics: [general use](#), [fulltext search](#), [HTTP REST API](#).

In the example above, the `indexed_content` field corresponds to the composite index over two fields: `title` and `content` (this index is generated when the namespace is created by the request from the preprocessor). [Text of the search query](#) starts with `@title^3, content^1` that means that the `title` field of the composite index has triple priority (i.e. weighting factor of 3), and the `content` field has normal priority (i.e. weight coefficient equals to 1). Also the example uses the `snippet()` [select function](#) to highlight the text that matches the query and to cut off excess.

If you use self-hosted instance of Reindexer, you may need to configure a proxy to append [CORS](#) headers to HTTP API responses.

## 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

[pypi](#) v2.0.0

[GitHub](#) v2.0.0

## Replace text for Foliant

Preprocessor for simple search and replace in Markdown sources with support of regular expressions.

### 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:

```
1 preprocessors:
2 - replace:
3 dictionary:
4 Mike: Michael
5 Sam: Samuel
6 Tim: Timoel
7 re_dictionary:
8 '!\\[\\]\\((.+?)\\)': '!Figure](\\1)'
```

**dictionary** YAML mapping where key is string to replace, value is the replacement string.

**re\_dictionary** YAML mapping where key is [Python regular expression](#) pattern, value is the replacement string.

## Usage

Fill up the `dictionary` or/and `re_dictionary` in preprocessor options and the keys will be replaced with values.

For example, if you wish that all images without title in your Markdown sources were titled “Figure”, use the following config:

```
1 preprocessors:
2 - replace:
3 re_dictionary:
4 '!\\[\\]\\((.+?)\\)': '!Figure](\\1)'
```

## 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:

## 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: ""
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 preprocessor applies to all targets.

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

## RunCommands

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

### Installation

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

### Usage

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

```
1 preprocessors:
2 - runcommands:
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}}
```

```
19 ``diff
20 {{diff}}
21 ...
22 {{endcommits}}
23 targets: []
```

**repo\_path** Path to the locally cloned copy of the Git repository that the current Foliант 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 `&quot;`.

**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 preprocessor 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](#) v1.0.12

[GitHub](#) v1.0.12

## 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
superlinks in the list
3 - ...
4 - superlinks
5 - ... # following preprocessors must be defined
AFTER superlinks in the list
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.

The body of the `link` tag is your link caption. If the body is empty, SuperLinks will try to guess the right caption:

- referenced title for links by title,
- meta section title for links by meta section,
- heading title for links by CustomIDs,
- title from config or first heading title in the file for links to file,
- anchor name for links by anchors.

## 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>
```

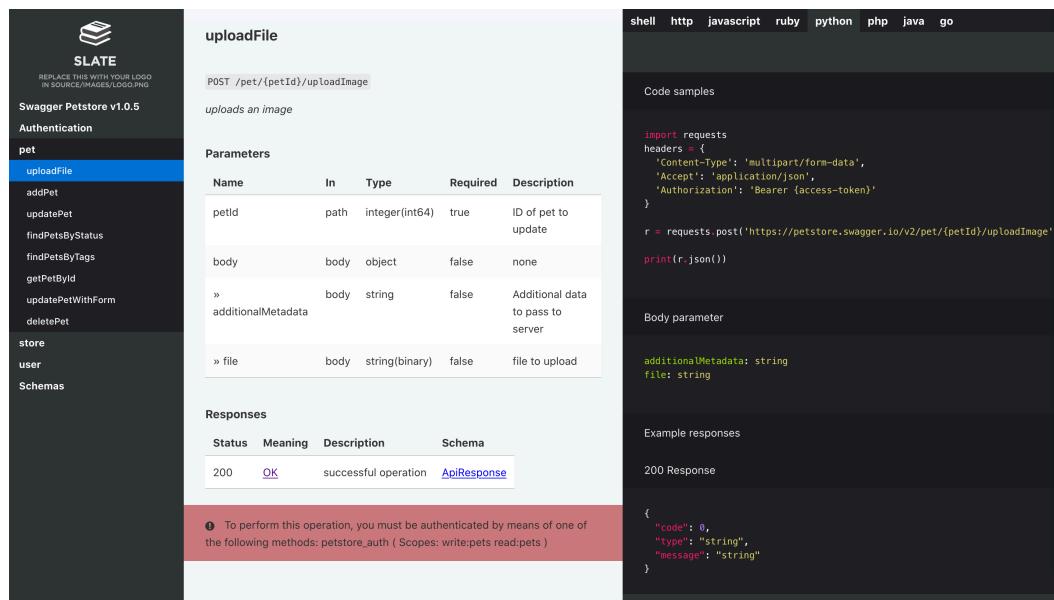
**Supported Backends:**

Backend	Support
aglio	□ YES
pandoc	□ YES
mdtopdf	□ YES
mkdocs	□ YES
slate	□ YES
confluence	□ YES

## SwaggerDoc

[pypi](#) v1.2.4

## 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
 documentation of our API.
4
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](#) v1.0.6

[GitHub](#) v1.0.6

# 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>
```

## Built-in variables

There are some variables that are available in your template by default:

- `_foliant_context` – dictionary with all user-defined variables, from tag parameters, `context` or `ext_context` variables,
- `_foliant_vars` – dictionary with all variables mentioned below (in case of name collisions),
- `meta` – dictionary with current chapter's metadata, details in the next chapter,
- `meta_object` – project's meta object, details in the next chapter,
- `config` – Foliant project config,
- `target` – current target,
- `backend` – current backend.

## 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.**

Pro tips

#### 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']`).

# Testrail

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

Important notice! We have some problems with displaying an exclamation mark in the image links, so they are replaced with `(leading_exclamation_mark_here)` phrase in the text.

## 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
 \\ Required
```

```

4 testrail_login: username
 \\ Required
5 testrail_pass: !env TESTRAIL_PASS
 \\ Required
6 project_id: 35
 \\ Required
7 suite_ids:
 \\ Optional
8 section_ids:
 \\ Optional
9 exclude_suite_ids:
 \\ Optional
10 exclude_section_ids:
 \\ Optional
11 exclude_case_ids:
 \\ Optional
12 filename: test_cases.md
 \\ Optional
13 rewrite_src_files: false
 \\ Optional
14 template_folder: case_templates
 \\ Optional
15 img_folder: testrail_imgs
 \\ Optional
16 move_imgs_from_text: false
 \\ Optional
17 section_header: Testing program
 \\ Recommended
18 std_table_header: Table with testing results
 \\ Recommended
19 std_table_column_headers: №; Priority; Platform; ID;
 Test case name; Result; Comment \\ Recommended
20 add_std_table: true
 \\ Optional
21 add_suite_headers: true
 \\ Optional

```

```
22 add_section_headers: true
23 \\" Optional
24 add_case_id_to_case_header: false
25 \\" Optional
26 add_case_id_to_std_table: false
27 \\" Optional
28 multi_param_name:
29 \\" Optional
30 multi_param_select:
31 \\" Optional
32 multi_param_select_type: any
33 \\" Optional
34 add_cases_without_multi_param: true
35 \\" Optional
36 add_multi_param_to_case_header: false
37 \\" Optional
38 add_multi_param_to_std_table: false
39 \\" Optional
40 checkbox_param_name:
41 \\" Optional
42 checkbox_param_select_type: checked
43 \\" Optional
44 choose_priorities:
45 \\" Optional
46 add_priority_to_case_header: false
47 \\" Optional
48 add_priority_to_std_table: false
49 \\" Optional
50 resolve_urls: true
51 \\" Optional
52 screenshots_url: https://gitlab_repository.url
53 \\" Optional
54 img_ext: .png
55 \\" Optional
56 print_case_structure: true
57 \\" For debugging
```

**testrail\_url** URL of TestRail deployed.  
**testrail\_login** Your TestRail username.  
**testrail\_pass** Your TestRail password.

It is not secure to store plain text passwords in your config files. We recommend to use [environment variables](#) to supply passwords.

**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 its 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 its URL, for example http://testrails.url/index.php?/suites/view/124&group\_by=cases:section\_id&group\_order=asc&group\_id=3926 <-.  
**exclude\_suite\_ids** You can exclude any suites (even stated in suite\_ids) from the document.  
**exclude\_section\_ids** The same with the sections.  
**exclude\_case\_ids** And the same with the cases.  
**filename** Path to the test cases file. It should be added to project chapters in foliant.yml. Default: test\_cases.md. For example:

```
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 - appendix.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 {
12 'content': 'Enter mobile phone number.',
13 'expected': '- Entered phone number '
14 'is visible in the form field.'
15 },
16 {
17 'content': 'Press OK button.',
18 'expected': '- SMS with registration code '
19 'received.\n'},
20 {
21 'content': 'SMS with registration code received.'},
22 {
23 'content': 'SMS with registration code received.'}
24],
25 'custom_test_androidtv': None,
26 'custom_test_appletv': None,
27 'custom_test_smarttv': 'None',
28 'custom_tp': True,
29 'estimate': None,
30 'estimate_forecast': None,
31 'id': 15940,
32 'milestone_id': None,
33 'priority_id': 4,
34 'refs': None,
35 'section_id': 3441,
36 'suite_id': 101,
37 'template_id': 7,
38 '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': '('
19 leading_exclamation_mark_here)[Image caption](testrail_imgs
20 /123.png)'},
21 ...
22]
23 }
```

**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 (leading_exclamation_mark_here)[Button](index.php?/
attachments/get/740)
6
7 Result 1:
8
9 Dialog box will opened:
10
11 (leading_exclamation_mark_here)[Dialog box](index.php?/
attachments/get/741)
12
13 Step 2:
14
15 Select option:
16
17 (leading_exclamation_mark_here)[List of options](index.php?/
attachments/get/742)
18
19 Result 2:
20
21 Option selected:
22
23 (leading_exclamation_mark_here)[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': '(leading_exclamation_mark_here)[
3 Button](testrail_imgs/740.png)'},
4 {'id': '741', 'link': '(leading_exclamation_mark_here)[
5 Dialog box](testrail_imgs/741.png)'},
6 {'id': '742', 'link': '(leading_exclamation_mark_here)[
7 List of options](testrail_imgs/742.png)'},
8 {'id': '743', 'link': '(leading_exclamation_mark_here)[
9 Result](testrail_imgs/743.png)'}
10]
```

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
 | result | Passed | Comment |
5 +=====+=====+=====+

6 | 1 | Press the button | Dialog box
 | will opened: | | |
7 | | | |
 | | | |
8 | | [Button](#740) | [Dialog box
 |](#741) | | |
9 | | | |
 | | | |
10 +---+-----+-----+

11 | 2 | Select option: | Option
 | selected: | | |
12 | | | |
 | | | |
13 | | [List of options](#742) | [Result
 |](#743) | | |
14 +---+-----+-----+

15
16 *Images:
17
18 <anchor>740</anchor>
19
20 (leading_exclamation_mark_here)[Button](testrail_imgs/740.
 png)
```

```

21
22 <anchor>741</anchor>
23
24 (leading_exclamation_mark_here)[Dialog box](testrail_imgs
 /741.png)
25
26 <anchor>742</anchor>
27
28 (leading_exclamation_mark_here)[List of options](
 testrail_imgs/742.png)
29
30 <anchor>743</anchor>
31
32 (leading_exclamation_mark_here)[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 №; 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 with checked field will be downloaded,
- unchecked – only cases with 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 |
4 changelog.md|
5 foliant|
6 |
7 preprocessors|
8 |
9 LICENSE|
10 README.md|
11 setup.py
12
13 2 directories, 5 files
```

## Src

### Installation

To enable the `src` command, install MultiProject extension:

### 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]
 [-n] [-d] SUBPATH
```

<sup>3</sup>

```
4 Generate the config file to build the project subset from
 SUBPATH.
```

```

5
6 positional arguments:
7 SUBPATH Path to the subset of the Foliant
8 project
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 foliant.yml
4 src
5 group_1
6 product_1
7 feature_1
8 foliant.yml
9 index.md
10 product_2

```

```

11 |
12 | └── foliant.yml
13 |
14 | └── main.md
15 └── group_2
16 ├── foliant.yml
17 └── intro.md

```

The command `foliant subset group_1/product_1/feautre_1` will merge the files `./src/group_1/product_1/feautre_1/foliant.yml` and `./foliant.yml`, and write the result into the file `./foliant.yml.subset`.

After that you may use the command like the following to build your Foliant project:

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

Let's look at some examples.

The content of the common (default) file `./foliant.yml`:

```

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 v0.2.1

GitHub v0.2.1

### 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
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.

## DownloadFile

[pypi v1.0.1](#)

[GitHub v1.0.1](#)

## DownloadFile Extension

DownloadFile is a configuration extension for Foliant which downloads external files to use in your project.

It also resolves `!download` YAML tag in the project config and inside XML-tags parameters.

### Installation

```
$ pip install foliantcontrib.downloadfile
```

### Usage

To configure DownloadFile add the following section to your `foliant.yml` file:

```
1 downloadfile:
2 fail_fast: true
3 ignore_ssl_errors: false
4 queue:
5 - url: https://example.com/image.png # required
6 save_to: images/img1.png
7 login: john
8 password: qwerty123
9 - ...
```

**`fail_fast`** When `true`, build will be stopped if any file can't be downloaded. If `false` – unavailable files will be just skipped. Doesn't affect `!download` tag, this one will always break the build on errors. Default: `true`.

**`ignore_ssl_errors`** Switch to `true` to skip SSL certificate check. Default: `false`.

**`queue`** list of files to download. Each file is represented by a dictionary with the following fields:

**`url (required)`** URL to the file which should be downloaded.

**`save_to`** path where the downloaded file should be saved, relative to the project root. If not supplied, file will be saved in the project root with the name from url.

**`login`** login for basic authentication.

**`password`** password for basic authentication.

**Warning:** don't store plain text passwords in foliant.yml. Use [environment variables](#).

`!download` YAML tag

Another way to use DownloadFile is by specifying `!download` YAML tag. This is the quickest and the simplest way, but it comes with a few disadvantages.

Insert the `!download` tag, followed by file URL, in any place in foliant.yml or tag parameters, where file path is expected:

```
1 preprocessors:
2 - swaggerdoc:
3 spec_path: !download https://example.com/swagger.
4 json
5 mode: widdershins
```

```
1 Generated template:
```

```
2
3 <template engine="jinja2" ext_context="!download https://
4 example.com/mycontext.yml">
5 </template>
```

The downloaded file will be saved in the `.downloadfilecache` directory under a hashed name, and the `!download` tag will be replaced by absolute path to this file.

The cons of this method are that you can't change the saved file path nor other parameters. Also if you reference the same file twice with `!download` tag, it will be downloaded two times.

## 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
```

## YAMLInclude

[pypi](#) v1.0.1

[GitHub](#) v1.0.1

# YAMLInclude Extension

YAMLInclude is a configuration extension for Foliant to include parts of configuration from YAML-files.

It resolves `!include` YAML tag in the project config and inside XML-tags parameters.

## Installation

```
$ pip install foliantcontrib.yaml_include
```

## Usage

The syntax of the `!include` YAML tag is:

```
!include <file>[#<key>]
```

Where `file` may be

- path to local file in Foliant project root,
- direct link to a file on remote server.

An optional `#<key>` part is used to get a key from the mapping stored inside `<file>`.

### Including a local file

Config example:

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

In this example the `chapters.yml` file should be placed in your Foliant project root.

if the contents of `chapters.yml` are as follows:

```
1 # chapters.yml
2
3 - index.md
4 - description.md
```

then the resulting config after applying the extension will be:

```
1 chapters:
2 - index.md
3 - description.md
```

## Including part of a local file

Config example:

```
1 chapters: !include chapters.yml#chapters_for_pdf
```

In this example the `chapters.yml` file should be placed in your Foliant project root.

if the contents of `chapters.yml` are as follows:

```
1 # chapters.yml
2
3 chapters_for_site:
4 - index_site.md
5 - description_site.md
6 chapters_for_pdf:
7 - index.md
8 - description.md
```

then the resulting config after applying the extension will be:

```
1 chapters:
2 - index.md
3 - description.md
```

## Including a remote file

Config example:

```
1 chapters: !include http://example.com/chapters.yml
```

In this example the `chapters.yml` file is stored on the website `http://example.com/`.

if the contents of `chapters.yml` are as follows:

```
1 # chapters.yml
2
3 - index.md
4 - description.md
```

then the resulting config after applying the extension will be:

```
1 chapters:
2 - index.md
3 - description.md
```

### Including part of a remote file

Config example:

```
chapters: !include http://example.com/chapters.yml#
chapters_for_pdf
```

In this example the `chapters.yml` file is stored on the website `http://example.com/`.

if the contents of `chapters.yml` are as follows:

```
1 # chapters.yml
2
3 chapters_for_site:
4 - index_site.md
5 - description_site.md
6 chapters_for_pdf:
7 - index.md
8 - description.md
```

then the resulting config after applying the extension will be:

```
1 chapters:
2 - index.md
3 - description.md
```

# History of Releases

Here is the single linear history of releases of Foliant and its extensions. It's also available as an [RSS feed](#).

## [2021-12-12] [foliantcontrib.confluence](#) 0.6.20

- Support for Confluence Cloud option to remove HTML formatting.
- Page URL is now taken from the properties.
- Article change is now detected by article body and title hash, stored in page properties.

## [2021-12-12] [foliantcontrib.replace](#) 2.0.0

- Preprocessor rewritten.

## [2021-12-12] [foliantcontrib.superlinks](#) 1.0.12

- Anchors added to beginning of files are not random anymore.

## [2021-10-07] [foliantcontrib.pandoc](#) 1.1.2

- Add `odt` and `epub` targets basic support.

## [2021-08-17] [foliantcontrib.dbdoc](#) 0.1.8

- DBMS python connectors are only imported on use.

## [2021-08-04] [foliantcontrib.argdown](#) 0.1.1

- Embed dot and graphml formats with `as_image=False`.

## [2021-08-03] [foliantcontrib.argdown](#) 0.1.0

- Initial release

## [2021-08-03] [foliantcontrib.superlinks](#) 1.0.11

- Fix imports.

## [2021-08-02] [foliantcontrib.bpmn](#) 1.0.1

- Initial release

## [2021-08-02] [foliantcontrib.pgsqldoc](#) 1.1.7

- New utils module

## [2021-08-02] [foliantcontrib.apilinks](#) 1.2.6

- New utils module

## [2021-08-02] [foliantcontrib.utils](#) 1.0.3

- PreprocessorExt: add `debug_msg` param to `_warning` method.

## [2021-07-21] [foliantcontrib.pandoc](#) 1.1.1

- Passing metadata parameters via config.
- Fix: images didn't render during separate sections build.

## [2021-07-21] [foliantcontrib.meta](#) 1.3.3

- New utils module.

## [2021-07-21] [foliantcontrib.csvtables](#) 1.0.2

- New utils module.

## [2021-07-21] [foliantcontrib.mermaid](#) 1.0.2

- New utils module.

## [2021-07-21] [foliantcontrib.ramldoc](#) 1.0.2

- New utils module.

## [2021-07-21] [foliantcontrib.admonitions](#) 1.0.1

- New utils module.

## [2021-07-21] [foliantcontrib.metagraph](#) 0.1.3

- New utils module.

## [2021-07-21] [foliantcontrib.swaggerdoc](#) 1.2.4

- New utils module.

## [2021-07-20] [foliantcontrib.apireferences](#) 1.0.2

- New utils module.

## [2021-07-20] [foliantcontrib.dbdoc](#) 0.1.7

- New utils module.

## [2021-07-20] [foliantcontrib.dbmldoc](#) 0.3.1

- New utils module.
- Update PyDBML parser to 0.4.1.

## [2021-07-20] [foliantcontrib.plantuml](#) 1.0.10

- New utils module.

## [2021-07-20] [foliantcontrib.graphviz](#) 1.1.5

- New utils module.
- Output syntax errors in stdout.

## [2021-07-20] [foliantcontrib.templateparser](#) 1.0.6

- New utils module.

## [2021-07-20] [foliantcontrib.alt\\_structure](#) 0.2.1

- New utils module.

## [2021-07-20] [foliantcontrib.anchors](#) 1.0.7

- New utils module.

## [2021-07-20] [foliantcontrib.confluence](#) 0.6.19

- New utils module.

## [2021-07-19] [foliantcontrib.superlinks](#) 1.0.10

- New utils module.

## [2021-07-15] [foliantcontrib.utils](#) 1.0.2

- Fix error in initial values for combined options classes.
- Combined options classes now clone input dicts and lists.
- Combined options: fix validate\_in.
- Header Anchors: fix slate id generator.
- PreprocessorExt: allow\_fail now supports methods without args.

## [2021-07-14] [foliantcontrib.utils](#) 1.0.1

- All utils from separate repositories now reside here.

## [2021-06-18] [foliantcontrib.plantuml](#) 1.0.9

- Diagrams with same options now generate in single PlantUML instance.
- Error tracebacks now are shown in console, [error images are not generated](#).
- Markdown image tags for broken diagrams are not inserted so they won't crash the build of the project.

## [2021-05-20] [foliantcontrib.dbmldoc](#) 0.3.0

- Update PyDBML parser to 0.4.0. This breaks backward compatibility with previous versions (reference cols are now lists).

## [2021-05-07] [foliantcontrib.multiproject](#) 1.0.15

- Fix crash caused by YAML-tags in subproject config.

## [2021-05-07] [foliantcontrib.yaml\\_include](#) 1.0.1

- Better logging.
- Improved paths handling.

## [2021-05-06] [foliantcontrib.downloadfile](#) 1.0.1

- Better work with multiproject.
- `fail_fast` parameter.

## [2021-05-04] [foliantcontrib.downloadfile](#) 1.0.0

- Initial release.

## [2021-04-20] [foliantcontrib.anchors](#) 1.0.6

- Fix: missing dependency in setup.py

## [2021-04-01] [foliantcontrib.confluence](#) 0.6.18

- Fix: external images didn't work.

## [2021-03-11] [foliantcontrib.apilinks](#) 1.2.5

- APILinks is now deprecated. Use APIReferences instead.

## [2021-03-11] [foliantcontrib.apireferences](#) 1.0.1

- Better logging and output.
- New: `warning_level` param.

## [2021-03-10] [foliantcontrib.confluence](#) 0.6.17

- Fix: parent\_id param didn't work.

## [2021-03-10] [foliantcontrib.yaml\\_include](#) 1.0.0

- Initial release.

## [2021-03-09] [foliantcontrib.apipreferences](#) 1.0.0

- Initial release.

## [2021-02-18] [foliantcontrib.confluence](#) 0.6.16

- New: attaching arbitrary files with help of `attachments` parameter.
- New: supply attachments implicitly using `ac:image` tag, without mentioning them in `attachments` parameter.
- Attachments and images which were referenced several times on a page will now only be uploaded once.
- Allow `!path`, `!project_path` modifiers inside `ac:attachment` param for `ac:link`, `ac:image`.

## [2021-02-03] [foliantcontrib.apilinks](#) 1.2.4

- Better warning when there's error in API configuration
- Trailing slash is not enforced in generated urls. Previously didn't work with \*.html sites

## [2021-02-01] [foliantcontrib.plantuml](#) 1.0.8

- Config options now can be overriden in tag options.
- Add `as_image` option, which allows (when `false`) to insert svg-code instead of image into the document.

## [2021-01-25] [foliantcontrib.dbdoc](#) 0.1.6

- Templates now support imports and includes from the same folder.

- Fix: Remove TEXT\_VC field from Oracle views query, which is absent on some versions.

## [2021-01-22] [foliantcontrib.apilinks](#) 1.2.3

- Added options `login` and `password` for basic authentication on API sites.

## [2020-12-18] [foliantcontrib.confluence](#) 0.6.15

- New: [experimental] `raw_confluence` tags are now not necessary for `ac:...` tags, they are escaped automatically.
- New: supply images with additional parameters using `ac:image` tag.
- New: `verify_ssl` parameter.

## [2020-12-04] [foliantcontrib.templateparser](#) 1.0.5

- Config, backend name and target are now available under `config`, `backend` and `target` variables.
- All the above variables along with `meta` and `meta_object` are now moved from `_foliant_context` into `_foliant_vars` variable.
- Fix: external context was overriding meta variables.

## [2020-12-03] [foliantcontrib.replace](#) 1.0.5

- Bug fixed with several replaceable items in one string.

## [2020-11-27] [foliantcontrib.superlinks](#) 1.0.9

- Fix: BOF anchors bug
- Fix: common paths processing

## [2020-11-17] [foliantcontrib.customids](#) 1.0.7

- Styles are now inserted after YAML Front Matter if that is present.

## [2020-11-16] [foliantcontrib.utils](#) 1.0.0

- Initial release.

## [2020-11-10] [foliantcontrib.dbdoc](#) 0.1.5

- New: trusted\_connection option for Microsoft SQL Server

## [2020-11-02] [foliantcontrib.apilinks](#) 1.2.2

- Fix: Endpoint prefix was ignored in swagger and redoc site backends.
- Better logging.

## [2020-10-28] [foliantcontrib.superlinks](#) 1.0.8

- Fix: proper relative path generation for links.
- Fix: multiple issues when !path modifier is used in the link tag.
- BOF anchors now won't be added for mkdocs backend.

## [2020-10-20] [foliantcontrib.pandoc](#) 1.1.0

- Option to build separate chapters (sections) into separate files.

## [2020-10-16] [foliantcontrib.confluence](#) 0.6.14

- Add code blocks processing for Confluence preprocessor.

## [2020-10-14] [foliantcontrib.graphviz](#) 1.1.4

- Fix: issue with MkDocs: raw svgs are now wrapped in div tag.

## [2020-10-08] [foliantcontrib.superlinks](#) 1.0.7

- If tag body is empty, superlinks will try to guess the right caption of the link:
  - referenced title for links by title,
  - meta section title for links by meta section,
  - heading title for links by CustomIDs,
  - title from config or first heading title in the file for links to file,
  - anchor name for links by anchors.

## [2020-10-07] [foliantcontrib.apilinks](#) 1.2.1

- Renamed spec\_url to spec because it may also be a path to local file,

- Improved swagger.json parsing
- Added Redoc support ( `redoc` site backend)

## [2020-10-06] [foliantcontrib.apilinks](#) 1.2.0

- Added Swagger UI support,
- Anchors are now generated properly, with `header_anchors` tool. Added `site_backend` optional param to determine for which backend the anchors should be generated.

## [2020-09-16] [foliantcontrib.archeme](#) 1.0.3

- Add `config_concat` option.

## [2020-09-07] [foliantcontrib.reindexer](#) 1.0.1

- Add the `fulltext_config` option.
- Fix web application example: mention the `content` field, not only `title`, in the query filter settings.

## [2020-08-26] [foliantcontrib.superlinks](#) 1.0.6

- Improved Confluence links: if section is not uploaded to Confluence, reference to overall project (if it is uploaded to Confluence).

## [2020-08-25] [foliantcontrib.superlinks](#) 1.0.5

- New: added Confluence backend support.
- Fix: links were corrupted when customids were used.
- Fix: several other bug fixes and optimizations.

## [2020-08-24] [foliantcontrib.swaggerdoc](#) 1.2.3

- Fix: build failed when spec referenced to other files with `$ref`.

## [2020-08-21] [foliantcontrib.confluence](#) 0.6.13

- Fix: cache dir for preprocessor was not created

## [2020-08-21] [foliantcontrib.dbdoc](#) 0.1.4

- New: Added support for MySQL

## [2020-08-20] [foliantcontrib.dbdoc](#) 0.1.3

- New: Added support for Microsoft SQL Server

## [2020-07-31] [foliantcontrib.escapecode](#) 1.0.4

- Addition to normalization: remove BOM.

## [2020-07-29] [foliantcontrib.includes](#) 1.1.13

- When getting the included content by URL, take into account the `charset` parameter of the `Content-Type` response header field.

## [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.

## [2020-04-23] [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.

## [2020-04-01] [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

## [2020-03-12] [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

## [2020-02-06] [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 independently 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.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.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: Cliar, 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 Cliar: 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

## [2019-10-01] [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.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

## [2019-08-15] [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` preprocessors must be installed.

## [2019-07-30] [foliantcontrib.includes](#) 1.1.1

- Support `escape_code` config option. Require Foliant 1.0.10 and `escapecode` processor 1.0.0.
- Process `sethead` recursively.

## [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.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

## [2019-05-17] [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.

## [2019-05-14] [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](#) 1.0.5

- Anchor headings conflicts are now more accurate, because for flat backends are checked for all chapters.
- Anchors with illegal characters (list in readme) now will be removed.
- Preprocessor now can also work with custom ids.

## [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

## [2019-03-14] [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

## [2019-02-12] [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

- Ninja 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.