Fabricius

Release 0.2.0

Predeactor

Dec 04, 2023

GUIDES

Fabricius: Python templates renderer

Fabricius is a tool that allows you to render files template & projects template.

Key features of Fabricius:

- (Will) Ship with its own project templating solution
- Supports CookieCutter templates
- Extendable with observers (AKA signals)
- User-friendly API

CHAPTER

ONE

INSTALLATION

The primary requirement of Fabricius is Python. It must be a version equal to or greater to Python `3.10`.

You can install Fabricius using `pip`, the Python's package manager (It comes bundled with Python). Install Fabricius using the following command:

```
pip install Fabricius
```

Note: Typically, Fabricius should be installed globally on your system (As you shouldn't need it in a specific project, it's a tool). As such, Windows might tell you to add the `--user` option, if so, try doing `pip install fabricius --user`!

CHAPTER

TWO

GUIDES

Important: Guides are not ready yet!

Fabricius need more time to get ready! While we're working on the documentation too, Fabricius is **not ready**! Guides (for now) are here to show you how Fabricius can work and how you should expect things to work out.

2.1 Guide: Create your Forge file

Important: The forge file is still not supported in Fabricius. This guide will just show you what to expect out of it.

The Forge file is a file created in either your repo or in a template that allows you to define what Fabricius will do. Basically, it's a configuration file, like the *cookiecutter.json* for CookieCutter. The difference with other tools is that we use a Python file to allows you more customization, with this approach, you can not only define how you want to create your template(s), but also:

1. Run some code you've made yourself instead of launching Fabricius (It's up to you to create files! Perfect for use with the `Generator`!)
2. Add plugins when running Fabricius
3. Have a fully type-hinted/type-safe config file

2.2 Guide: Rendering files & templates

Oh hey, nice finding! Sadly, this is a work in progress, come check this out later :)

2.3 Guide: Rendering CookieCutter templates

Fabricius ships with the ability to process CookieCutter templates (Or so called, cookiecutters by themselves). This mean that all of the work you've already done using CookieCutter **is 100% supported** in Fabricius! (yes! hooks work too!)

2.3.1 Using the CLI

TBD

2.3.2 Using the API

You can also use Fabricius's API to generate your CookieCutter template.

```
from fabricius.readers.cookiecutter.setup import setup, run

def main():
    template_path = "path/to/template"
    output_path = "path/to/output"
    template = setup(template_path, output_path)

    # This method has been specially created for CookieCutter's solution.
    # See below for explanations.
    run(template)
```

2.3.3 API

The following functions are available as part of the little API you can use to generate *fabricius.models.template.Template* objects from a CookieCutter repo.

The `setup` function will:

1. Get the template path
2. Create the Template object and add the extensions, if the project templates specify any others, add them too.
3. Add `_template`, `_repo_dir` and `_output_dir` to the context
4. Obtain the questions in the project template and begin to ask to the users those questions.
5. Once all answered, add the extra context the user's default context, then add the answers to the final context.
6. Obtain all the files that must be rendered/copied, and add them to the Template object, and push the data to the Template object.
7. Connect the hooks to the `before_template_commit` and `after_template_commit` signals, then return the Template object.

While you can just simply do *Template.commit()*, there is a few things to considerate first since you're rendering a CookieCutter project, and not a Fabricius one. Thus, we have made the `run` function to handle a few edge cases that could happens with CookieCutter.

The `run` function will:

1. First attempt to commit the project
2. If fail, due to a file that already exist, ask the user if overwriting files should be used.
3. If fail, due to a hook failing, see if the exception gives an exit code, if it does, exit using the exit code, if not, print the exception and exit.
4. Return the list of file commit result.

fabricius.readers.cookiecutter.setup.**setup**(*base_folder: str | os.PathLike[str] | pathlib.Path*, *output_folder: str | os.PathLike[str] | pathlib.Path*, ***, *extra_context: dict[str, Any] | None = None*, *no_prompt: bool = False*) → *Template*[type[*fabricius.renderers.jinja_renderer.JinjaRenderer*]]

Setup a template that will be able to be ran once created.

Parameters

- **base_folder** (*PathStrOrPath*) – The folder where the template is located. (Choose the folder where the `cookiecutter.json` is located, not the template itself)
- **output_folder** (*PathStrOrPath*) – The folder where the template/files will be created once rendered.
- **extra_context** (*Data*, optional) – Any extra context to pass to the template. It will override the user's prompt.
- **no_prompt** (*bool, optional*) – If set to True, no questions will be asked to the user. By default False

Returns

The Template that has been generated. It is ready to be committed, and everything has been setup.

Return type

Type of *fabricius.models.template.Template*

Raises

fabricius.exceptions.TemplateError – Exception raised when there's an issue with the template that is most probably due to the template's misconception.

fabricius.readers.cookiecutter.setup.**run**(*template: Template[type[*fabricius.renderers.jinja_renderer.JinjaRenderer*]]*) → list[*fabricius.types.FileCommitResult*]

Run the CookieCutter template generated using `setup()`

Parameters

template (Type of *fabricius.models.template.Template*) – The template to render.

CHAPTER

THREE

API

Careful here, commander!

This section is reserved for the peoples that are interested to use more complex tools in order to better understand how Fabricius works behind the scene & use it themselves.

If you believe your use case is easy to tackle down, then you probably don't need to dig into the Fabricius's API.

3.1 Models

Models represents the objects used inside Fabricius, for example, the *File* model is an object that represent a file we're about to create.

In this page, you're able to visit the available methods in our models and freely uses them.

class fabricius.models.file.**File**(*name: str*, *extension: Optional[str] = None*)

The builder class to initialize a file template. The result (Through the *generate()* method) is the render of the file's content. You can "commit" the file to the disk to persist the file's content.

Parameters

- **name** (str) – The name of the file.
- **extension** (str) – The extension of the file, without dot, same as name="<name>.<extension>" (Where <name> and <extensions> are the arguments given to the class).

template_content: str | None

The content of the base template, if set.

name: str

The name of the file that will be generated.

state: Literal['pending', 'persisted']

The state of the file.

content: str | None

The template's content.

destination: pathlib.Path | None

The destination of the file, if set.

renderer: type[*fabricius.models.renderer.Renderer*]

The renderer to use to generate the file.

data: dict[str, Any]

The data that will be passed to the renderer.

compute_destination() → Path

Compute the destination of the file.

Raises
fabricius.exceptions.MissingRequiredValueError – If the object does not have the property "destination" set. (Use *to_directory()*)

Returns
The final path.

Return type
pathlib.Path

from_file(*path: str | pathlib.Path*) → Self

Read the content from a file template.

Raises
FileNotFoundError – If the file was not found.

Parameters
path (str or pathlib.Path) – The path of the file template.

from_content(*content: str*) → Self

Read the content from a string.

Parameters
content (str) – The template you want to format.

to_directory(*directory: str | os.PathLike[str] | pathlib.Path*) → Self

Set the directory where the file will be saved.

Raises
NotADirectory – The given path exists but is not a directory.

Parameters
directory (str or pathlib.Path) – Where the file will be stored. Does not include the file's name.

use_mustache() → Self

Use chevron (Mustache) to render the template.

use_string_template() → Self

Use string.Template to render the template.

use_jinja() → Self

Use Jinja2 to render the template.

with_renderer(*renderer: Type[*Renderer*]*) → Self

Use a custom renderer to render the template.

Parameters
renderer (Type of *fabricius.models.renderer.Renderer*) – The renderer to use to format the file. It must be not initialized.

with_data(*data: dict[str, Any]*, *, *overwrite: bool = True*) → Self

Add data to pass to the template.

Parameters

- **data** (*fabricius.types.Data*) – The data you want to pass to the template.
- **overwrite** (bool) – If the data that already exists should be deleted. If False, the new data will be added on top of the already existing data. Default to `True`.

fake() → Self

Set the file to fake the commit. This will ensure that the file does not get stored on the machine upon commit.

restore() → Self

Set the file to not fake the commit. This will ensure that the file gets stored on the machine upon commit.

Hint: This is the default behavior. It's only useful to use this method if you have used *fake()*.

---

generate() → str

Generate the file's content.

Raises
fabricius.exceptions.MissingRequiredValue – If no content to the file were added.

Returns
The final content of the file.

Return type
str

commit(*, *overwrite: bool = False*) → *FileCommitResult*

Save the file to the disk.

Parameters
overwrite (bool) – If a file exist at the given path, shall the overwrite parameter say if the file should be overwritten or not. Default to `False`.

:raises MissingRequiredValueError <fabricius.exceptions.MissingRequiredValueError>` : If a required value was not set. (Content or destination) :raises fabricius.exceptions.AlreadyCommittedError: If the file has already been saved to the disk. :raises FileExistsError: If the file already exists on the disk and `overwrite` is set to `False`.

This is different from *AlreadyCommittedError* because this indicates that the content of the file this generator was never actually saved. :raises OSError: The file's name is not valid for the OS.

Returns
A typed dict with information about the created file.

Return type
fabricius.types.FileCommitResult

class fabricius.models.signal.**Signal**(*, *func_hint: Optional[Callable[[_F], Any]] = None*)

The Listener is the base class used to create listeners of events.

listeners: `list[Callable[_F, Any]]`

The list of listeners that are subscribed to this signal.

connect(*listener: Callable[[_F], Any]*) → None

Connect a listener to this signal.

disconnect(*listener: Callable[[_F], Any]*) → None

Disconnect a listener to this signal.

send(**args: ~typing.~_F*, ***kwargs: ~typing.~_F*) → list[Any]

Sends the signal to all subscribed listeners.

class fabricius.models.renderer.**Renderer**(*data: dict[str, Any]*)

The Renderer is what translate and generates the output of templates. Core of the work.

You must subclass this class and override the *render()* method, if possible, also add a name.

name: `ClassVar[str | None] = None`

The name of the renderer, not necessary, but suggested to add.

data: `dict[str, Any]`

A dictionary that contains data passed by the users to pass inside the template.

abstract render(*content: str*) → str

This method will process a given string, the template input and return the processed template as a string too.

Parameters
content (str) – The template

Returns
The result of the processed template.

Return type
str

class fabricius.models.template.**Template**(*base_folder: str | os.PathLike[str] | pathlib.Path*, *renderer: RendererType*)

The *Template* class represent "a collection of files that, in its whole, represents a project"

The difference between the *Template* class and a collection of *File* is that a template assumes all of your files have the same properties. (Requires the same renderer, the same data, etc.)

Typically, a template only use one renderer, and shares the same data across the whole project template.

The *Template* will assist creating a project, while providing a similar interface of the *File* model.

Parameters

- **base_folder** (*fabricius.types.PathStrOrPath*) – Indication of where the template should be generated.
- **renderer** (Type of *fabricius.models.renderer.Renderer*) – The renderer to use with the template.

base_folder: `Path`

The folder where the template will be generated.

state: `Literal['pending', 'failed', 'persisted']`

The state of the template

files: `list[fabricius.models.file.File]`

The list of files that will be rendered when committing.

data: `dict[str, Any]`

The data to pass to the files.

renderer: `RendererType`

The renderer that will be used to generate the files.

add_file(*file:* File) → Self

add_files(*files:* *Iterable[*File*]*) → Self

push_data(*data:* *dict[str, Any]*) → Self

fake() → Self

restore() → Self

commit(*, *overwrite:* *bool* = *False*) → list[*fabricius.types.FileCommitResult*]

3.2 Renderers

The "Renderer" is a class that is created in order to generate the content of a template.

Fabricius ships many by default, you can use them, or create your own if you feel the need to.

```
from fabricius.renderers import Renderer

class MyRenderer(Renderer):

    # You must implement the "render" method, this will be called by Fabricius.
    def render(self, content: str):
        # Inside of the Renderer class, the "data" property is available.
        # This is where the data is stored.

        final_content = render_content(content=content, data=self.data)

        return final_content

renderer = MyRenderer({"name": "John"})
final_content = renderer.render("Hello {{ name }}")
print(final_content)
# Hello John
```

The following is the list of the available renderer packaged with Fabricius. It contains Python's str.format, string template, Mustache & Jinja.

Hint: If you're using the *File* object, you can use methods *File.use_jinja()* to set the renderer to one of Fabricius's available. To use your own Renderer, use *File.with_renderer()*.

---

class fabricius.renderers.**JinjaRenderer**(*data:* *dict[str, Any]*)

name: `ClassVar[str | None] = 'Jinja Template'`

The name of the renderer, not necessary, but suggested to add.

render(*content: str*) → str

This method will process a given string, the template input and return the processed template as a string too.

Parameters
content (str) – The template

Returns
The result of the processed template.

Return type
str

class fabricius.renderers.**ChevronRenderer**(*data: dict[str, Any]*)

name: ClassVar[str | None] = 'Chevron (Moustache)'

The name of the renderer, not necessary, but suggested to add.

render(*content: str*) → str

This method will process a given string, the template input and return the processed template as a string too.

Parameters
content (str) – The template

Returns
The result of the processed template.

Return type
str

class fabricius.renderers.**PythonFormatRenderer**(*data: dict[str, Any]*)

name: ClassVar[str | None] = 'Python str.format'

The name of the renderer, not necessary, but suggested to add.

render(*content: str*) → str

This method will process a given string, the template input and return the processed template as a string too.

Parameters
content (str) – The template

Returns
The result of the processed template.

Return type
str

class fabricius.renderers.**StringTemplateRenderer**(*data: dict[str, Any]*, *, *safe: bool = True*)

name: ClassVar[str | None] = 'Python string.Template'

The name of the renderer, not necessary, but suggested to add.

safe: bool

Indicate if the renderer should use `string.Template.safe_substitute()` or `string.Template.substitute()`

render(*content: str*) → str

This method will process a given string, the template input and return the processed template as a string too.

Parameters
content (str) – The template

Returns
The result of the processed template.

Return type
str

3.3 Types

This page is fairly short, but deserves to be shown.

This explains you the specific types available in Fabricius. They are widely used inside Fabricius in order to simply how the docs is rendered and to facilitate understanding of the library.

Fabricius types

fabricius.types.**Data**: TypeAlias = 'dict[str, typing.Any]'

Data is an alias that represents a dictionary of which every keys are string and their values of any types.

fabricius.types.**PathStrOrPath**: TypeAlias = 'str | os.PathLike[str] | pathlib.Path'

PathStrOrPath represents a path as a str or a pathlib.Path object. Used to help users either have an easy way to give their path.

class fabricius.types.**FileCommitResult**

A FileCommitResult is returned when a file was successfully saved. It returns its information after its creation.

name: str

The name of the file.

state: Literal['pending', 'persisted']

The state of the file. Should always be "persisted".

destination: Path

Where the file is located/has been saved.

data: dict[str, Any]

The data that has been passed during rendering.

template_content: str

The original content of the template.

content: str

The resulting content of the saved file.

fake: bool

If the file was faked. If faked, the file has not been saved to the disk.

3.4 Signals

Signals are observers. They permit you to run code when a specific action is happening.

There is a lot of signal that Fabricius raises so you can subscribe to any thing you'd like. For example, before committing a file, you can add a suffix to its name before it get committed.

```
from fabricius.app.signals import before_file_commit
from fabricius.models.file import File

on_file_commit(file: File):
    file.name = f"{file.name}.template"

before_file_commit.connect(on_file_commit)
```

Here is a list of the available signals raised in Fabricius.

`fabricius.app.signals.`**`before_file_commit = <fabricius.models.signal.Signal object>`**

A Signal called when a *`File`* is about to commit a file.

`fabricius.app.signals.`**`on_file_commit_fail = <fabricius.models.signal.Signal object>`**

A Signal called when a *`File`* had an exception occurring when committing a file.

`fabricius.app.signals.`**`after_file_commit = <fabricius.models.signal.Signal object>`**

A Signal called when a *`File`* has committed a file.

`fabricius.app.signals.`**`before_template_commit = <fabricius.models.signal.Signal object>`**

A Signal called when a *`Template`* is about to commit files.

`fabricius.app.signals.`**`after_template_commit = <fabricius.models.signal.Signal object>`**

A Signal called when a *`Template`* has committed all the files.

3.4.1 Create your own signals

You can create your own signal by creating a *`fabricius.models.signal.Signal`* object and letting it available in your project.

```
from fabricius.models.signal import Signal

my_signal = Signal()
```

While this is totally OK to go like this, you can also optionally type the *`.send()`*/*`.connect()`* methods by providing a function. Fabricius will extract the function's signature and use it to transfer the arguments into the signal's methods.

```
from fabricius.models.file import File
from fabricius.models.signal import Signal

def my_signal_hint(file: File):
    ...

my_signal = Signal(func_hint=my_signal_hint)

my_signal.send(File("test.py"))  # This is OK
my_signal.send()  # This is not! Your type checker will complain!
```

(continues on next page)

(continued from previous page)

```
# Good
def signal_receiver(file: File):
    ...
my_signal.connect(signal_receiver)

# Bad
def signal_receiver(receiving_file: File):
    ...
my_signal.connect(signal_receiver)

# This will raise a type error if the function's signature is altered
# (New, removed, renamed arguments, etc...)
```

3.5 Exceptions

Inside all of our logic behind it, we have created supplementary exceptions to explain what could have gone wrong either in your template, or potentially what could have badly happened.

exception fabricius.exceptions.**FabriciusError**(*error: object | None = None*)

An error was raised inside Fabricius.

All exceptions of the Fabricius library **must** subclass this exception.

exception fabricius.exceptions.**MissingRequiredValueError**(*instance: object*, *missing_value: str*)

Exception raised when a required value was not set inside an object.

Parameters

- **instance** (*object*) – The object that holds the missing value.
- **missing_value** (*str*) – The value that was not set.

exception fabricius.exceptions.**AlreadyCommittedError**(*file_name: str*)

A file has already been committed/persisted.

Parameters

file_name (*str*) – The file's name that has already been committed.

exception fabricius.exceptions.**ConflictError**(*instance: object*, *reason: str*)

Conflicting parameters between objects.

This mean that Fabricius cannot continue its work because there would be a conflict between one or two objects.

For example, this error can be raised when two files have the same name in the same destination folder.

Parameters

- **instance** (*object*) – The conflicting object.
- **reason** (*str*) – The reason for the conflict.

exception fabricius.exceptions.**TemplateError**(*template_name: str*, *reason: str*)

The template has an error that cannot be automatically handled by Fabricius.

Parameters

- **template_name** (*str*) – The name of the template that raised the error.

- **reason** (*str*) – The reason for the error.

PYTHON MODULE INDEX

f

INDEX

O

P

R

S

T

U

W