scaraplate

Release 0.5

Jul 23, 2023

Contents:

Documentation https://scaraplate.readthedocs.io/

Source Code https://github.com/rambler-digital-solutions/scaraplate

Issue Tracker https://github.com/rambler-digital-solutions/scaraplate/issues

PyPI https://pypi.org/project/scaraplate/

CHAPTER 1

Introduction

Scaraplate is a wrapper around cookiecutter which allows to repeatedly rollup project templates onto concrete projects.

Cookiecutter is a great tool which allows to create projects from templates. However, it lacks ability to update the already created projects from updated templates. Scaraplate provides a solution to this problem.

To use scaraplate, you would have to add a `scaraplate.yaml` file near the `cookiecutter.json` of your cookiecutter template. Then, to rollup the changes from the updated template, you will need to simply call this command (which can even be automated!):

```
scaraplate rollup ./path/to/template ./path/to/project --no-input
```

This allows to easily (and automatically) sync the projects from the template, greatly simplifying the unification of the projects' structure.

CI pipelines, code linting settings, test runners, directory structures, artifacts building tend to greatly vary between the projects. Once described in the template which is easy to rollup onto the specific projects, projects unification becomes a trivial task. Everything can be defined in the template just once and then regularly be synced onto your projects, see *Rollup Automation*.

CHAPTER 2

How it works

The `scaraplate rollup` command does the following:

1. Retrieve cookiecutter template variables from the previous rollup (see *Scaraplate Template*).
2. Create a temporary dir, apply *cookiecutter* command with the retrieved variables to create a new temporary project.
3. For each file in the temporary project, apply a *strategy* (see *Strategies*) which merges the file from the temporary project with the corresponding file in the target project.

Only the files which exist in the temporary project are touched by scaraplate in the target project.

The key component of scaraplate are the *strategies*.

Note that none of the strategies use git history or any git-like merging. In fact, scaraplate strategies make no assumptions about the code versioning system used by the target project. Instead, the merging between the files is defined solely by *strategies* which generate the output based on the two files and the settings in the `scaraplate.yaml`.

Scaraplate is quite extensible. Many parts are replaceable with custom implementations in Python.

CHAPTER 3

Quickstart

scaraplate requires Python 3.7 or newer.

Installation:

```
pip install scaraplate
```

Scaraplate also requires `git` to be installed in the system (see *Scaraplate Template*).

To get started with scaraplate, you need to:

1. Prepare a template (see *Scaraplate Template* and specifically *Scaraplate Example Template*).
2. Roll it up on your projects.

CHAPTER 4

Project Name

The project name is inspired by a bug which rolls up the brown balls of . . . well, stuff, (the template) everywhere (the projects).

scarab + tem**plate** = **scaraplate**

4.1 Scaraplate Template

Scaraplate uses cookiecutter under the hood, so the scaraplate template is a cookiecutter template with the following properties:

- There must be a `scaraplate.yaml` config in the root of the template dir (near the `cookiecutter.json`).
- The template dir must be a git repo (because some strategies might render URLs to the template project and HEAD commit, making it easy to find out what template was used to rollup, see *Template Git Remotes*).
- The cookiecutter's project dir must be called `project_dest`, i.e. the template must reside in the `{{cookiecutter.project_dest}}` directory.
- The template must contain a file which renders the current cookiecutter context. Scaraplate then reads that context to re-apply cookiecutter template on subsequent rollups (see *Cookiecutter Context Types*).

`scaraplate.yaml` contains:

- strategies (see *Strategies*),
- cookiecutter context type (see *Cookiecutter Context Types*),
- template git remote (see *Template Git Remotes*).

Note: Neither `scaraplate.yaml` nor `cookiecutter.json` would get to the target project. These two files exist only in the template repo. The files that would get to the target project are located in the inner `{{cookiecutter.project_dest}}` directory of the template repo.

`scaraplate rollup` has a `--no-input` switch which doesn't ask for cookiecutter context values. This can be used to automate rollups when the cookiecutter context is already present in the target project (i.e. `scaraplate rollup` has been applied before). But the first rollup should be done without the `--no-input` option, so the cookiecutter context values could be filled by hand interactively.

The arguments to the `scaraplate rollup` command must be local directories (i.e. the template git repo must be cloned manually, scaraplate doesn't support retrieving templates from git remote directly).

4.1.1 Scaraplate Example Template

We maintain an example template for a new Python project here: https://github.com/rambler-digital-solutions/scaraplate-example-template

You may use it as a starting point for creating your own scaraplate template. Of course it doesn't have to be for a Python project: the cookiecutter template might be for anything. A Python project is just an example.

Creating a new project from the template

```
$ git clone https://github.com/rambler-digital-solutions/scaraplate-example-template.
↪git
$ scaraplate rollup ./scaraplate-example-template ./myproject
`myproject1/.scaraplate.conf` file doesn't exist, continuing with an empty context...
`project_dest` must equal to "myproject"
project_dest [myproject]:
project_monorepo_name []:
python_package [myproject]:
metadata_name [myproject]:
metadata_author: Kostya Esmukov
metadata_author_email: kostya@esmukov.net
metadata_description: My example project
metadata_long_description [file: README.md]:
metadata_url [https://github.com/rambler-digital-solutions/myproject]:
coverage_fail_under [100]: 90
```

(continues on next page)

(continued from previous page)

```
mypy_enabled [1]:
Done!
$ tree -a myproject
myproject
├── .editorconfig
├── .gitignore
├── .scaraplate.conf
├── MANIFEST.in
├── Makefile
├── README.md
├── mypy.ini
├── setup.cfg
├── setup.py
├── src
│   └── myproject
│       └── __init__.py
└── tests
    ├── __init__.py
    └── test_metadata.py

3 directories, 12 files
```

The example template also contains a `project_monorepo_name` variable which simplifies creating subprojects in monorepos (e.g. a single git repository for multiple projects). In this case scaraplate should be applied to the inner projects:

```
$ scaraplate rollup ./scaraplate-example-template ./mymonorepo/innerproject
`mymonorepo/innerproject/.scaraplate.conf` file doesn't exist, continuing with an␣
↪empty context...
`project_dest` must equal to "innerproject"
project_dest [innerproject]:
project_monorepo_name []: mymonorepo
python_package [mymonorepo_innerproject]:
metadata_name [mymonorepo-innerproject]:
metadata_author: Kostya Esmukov
metadata_author_email: kostya@esmukov.net
metadata_description: My example project in a monorepo
metadata_long_description [file: README.md]:
metadata_url [https://github.com/rambler-digital-solutions/mymonorepo]:
coverage_fail_under [100]: 90
mypy_enabled [1]:
Done!
$ tree -a mymonorepo
mymonorepo
└── innerproject
    ├── .editorconfig
    ├── .gitignore
    ├── .scaraplate.conf
    ├── MANIFEST.in
    ├── Makefile
    ├── README.md
    ├── mypy.ini
    ├── setup.cfg
    ├── setup.py
    ├── src
    │   └── mymonorepo_innerproject
```

(continues on next page)

(continued from previous page)

```
    │       └── __init__.py
    └── tests
        ├── __init__.py
        └── test_metadata.py

4 directories, 12 files
```

Updating a project from the template

```
$ scaraplate rollup ./scaraplate-example-template ./myproject --no-input
Continuing with the following context from the `myproject/.scaraplate.conf` file:
{'_template': 'scaraplate-example-template',
 'coverage_fail_under': '90',
 'metadata_author': 'Kostya Esmukov',
 'metadata_author_email': 'kostya@esmukov.net',
 'metadata_description': 'My example project',
 'metadata_long_description': 'file: README.md',
 'metadata_name': 'myproject',
 'metadata_url': 'https://github.com/rambler-digital-solutions/myproject',
 'mypy_enabled': '1',
 'project_dest': 'myproject',
 'project_monorepo_name': '',
 'python_package': 'myproject'}
Done!
```

4.1.2 Cookiecutter Context Types

cookiecutter context are the variables specified in `cookiecutter.json`, which should be provided to cookiecutter to cut a project from the template.

The context should be generated by one of the files in the template, so scaraplate could read these variables and rollup the template automatically (i.e. without asking for these variables).

The default context reader is *ScaraplateConf*, but a custom one might be specified in `scaraplate.yaml` like this:

```
cookiecutter_context_type: scaraplate.cookiecutter.SetupCfg
```

class scaraplate.cookiecutter.**CookieCutterContext**(*target_path: pathlib.Path*)

Bases: abc.ABC

The abstract base class for retrieving cookiecutter context from the target project.

This class can be extended to provide a custom implementation of the context reader.

__init__(*target_path: pathlib.Path*) → None

Init the context reader.

read() → NewType.<locals>.new_type

Retrieve the context.

If the target file doesn't exist, `FileNotFoundError` must be raised.

If the file doesn't contain the context, an empty dict should be returned.

4.1.3 Built-in Cookiecutter Context Types

class scaraplate.cookiecutter.**ScaraplateConf**(*target_path: pathlib.Path*)
Bases: *scaraplate.cookiecutter.CookieCutterContext*

A default context reader which assumes that the cookiecutter template contains the following file named `.scaraplate.conf` in the root of the project dir:

```
[cookiecutter_context]
{%- for key, value in cookiecutter.items()|sort %}
{%- if key not in ('_output_dir',) %}
{{ key }} = {{ value }}
{%- endif %}
{%- endfor %}
```

Cookiecutter context would be rendered in the target project by this file, and this class is able to retrieve that context from it.

class scaraplate.cookiecutter.**SetupCfg**(*target_path: pathlib.Path*)
Bases: *scaraplate.cookiecutter.CookieCutterContext*

A context reader which retrieves the cookiecutter context from a section in `setup.cfg` file.

The `setup.cfg` file must be in the cookiecutter template and must contain the following section:

```
[tool:cookiecutter_context]
{%- for key, value in cookiecutter.items()|sort %}
{%- if key not in ('_output_dir',) %}
{{ key }} = {{ value }}
{%- endif %}
{%- endfor %}
```

4.1.4 Template Maintenance

Given that scaraplate provides ability to update the already created projects from the updated templates, it's worth discussing the maintenance of a scaraplate template.

Removing a template variable

Template variables could be used as *feature flags* to gradually introduce some changes in the templates which some target projects might not use (yet) by disabling the flag.

But once the migration is complete, you might want to remove the no longer needed variable.

Fortunately this is very simple: just stop using it in the template and remove it from `cookiecutter.json`. On the next `scaraplate rollup` the removed variable will be automatically removed from the *cookiecutter context file*.

Adding a new template variable

The process for adding a new variable is the same as for removing one: just add it to the `cookiecutter.json` and you can start using it in the template.

If the next `scaraplate rollup` is run with `--no-input`, the new variable will have the default value as specified in `cookiecutter.json`. If you need a different value, you have 2 options:

1. Run `scraplate rollup` without the `--no-input` flag so the value for the new variable could be asked interactively.

2. Manually add the value to the *cookiecutter context section* so the next `rollup` could pick it up.

Restructuring files

Scaraplate strategies intentionally don't provide support for anything more complex than a simple file-to-file change. It means that a scaraplate template cannot:

1. Delete or move files in the target project;
2. Take multiple files and union them.

The reason is simple: such operations are always the one-time ones so it is just easier to perform them manually once than to maintain that logic in the template.

4.1.5 Patterns

This section contains some patterns which might be helpful for creating and maintaining a scaraplate template.

Feature flags

Let's say you have a template which you have applied to dozens of your projects.

And now you want to start gradually introducing a new feature, let it be a new linter.

You probably would not want to start using the new thing everywhere at once. Instead, usually you start with one or two projects, gain experience and then start rolling it up on the other projects.

For that you can use template variables as feature flags. The *example template* contains a `mypy_enabled` variable which demonstrates this concept. Basically it is a regular cookiecutter variable, which can take different values in the target projects and thus affect the template by enabling or disabling the new feature.

Include files

Consider `Makefile`. On one hand, you would definitely want to have some make targets to come from the template; on the other hand, you might need to introduce custom make targets in some projects. Coming up with a scaraplate strategy which could merge such a file would be quite difficult.

Fortunately, `Makefile` allows to include other files. So the solution is quite trivial: have `Makefile` synced from the template (with the *`scaraplate.strategies.Overwrite`* strategy), and include a `Makefile.inc` file from there which will not be overwritten by the template. This concept is demonstrated in the *example template*.

Manual merging

Sometimes you need to merge some files which might be modified in the target projects and for which there's no suitable strategy yet. In this case you can use *`scaraplate.strategies.TemplateHash`* strategy as a temporary solution: it would overwrite the file each time a new git commit is added to the template, but keep the file unchanged since the last rollup of the same template commit.

The *example template* uses this approach for `setup.py`.

Create files conditionally

Cookiecutter hooks can be used to post-process the generated *temporary project*. For example, you might want to skip some files from the template depending on the variables.

The *example template* contains an example hook which deletes `mypy.ini` file when the `mypy_enabled` variable is not set to `1`.

4.2 Strategies

Strategies do the merging between the files from template and the target.

Strategies are specified in the `scaraplate.yaml` file located in the root of the template dir.

Sample `scaraplate.yaml` excerpt:

```
default_strategy: scaraplate.strategies.Overwrite
strategies_mapping:
  setup.py: scaraplate.strategies.TemplateHash
  src/*/__init__.py: scaraplate.strategies.IfMissing
  package.json:
    strategy: mypackage.mymodule.MyPackageJson
    config:
      my_key: True
  "src/{{ cookiecutter.myvariable }}.md": scaraplate.strategies.IfMissing
```

The strategy should be an importable Python class which implements *`Strategy`*.

`default_strategy` and `strategies_mapping` keys are the required ones.

The strategy value might be either a string (specifying a Python class), or a dict of two keys – `strategy` and `config`. The first form is just a shortcut for specifying a strategy with an empty config.

`config` would be passed to the Strategy's `__init__` which would be validated with the inner `Schema` class.

class scaraplate.strategies.**Strategy**

Bases: abc.ABC

The abstract base class for a scaraplate Strategy.

To implement and use a custom strategy, the following needs to be done:

1. Create a new Python class which implements *`Strategy`*
2. Override the inner `Schema` class if you need to configure your strategy from `scaraplate.yaml`.
3. Implement the `apply` method.

Assuming that the new strategy class is importable in the Python environment in which scaraplate is run, to use the strategy you need to specify it in `scaraplate.yaml`, e.g.

```
strategies_mapping:
  myfile.txt: mypackage.mymodule.MyStrategy
```

class Schema

An empty default schema which doesn't accept any parameters.

__init__(**, target_contents: Optional[BinaryIO], template_contents: BinaryIO, template_meta: scaraplate.template.TemplateMeta, config: Dict[str, Any]*) → None

Init the strategy.

Parameters

- **target_contents** – The file contents in the target project. `None` if the file doesn't exist.
- **template_contents** – The file contents from the template (after cookiecutter is applied).
- **template_meta** – Template metadata, see *scaraplate.template.TemplateMeta*.
- **config** – The strategy config from `scaraplate.yaml`. It is validated in this `__init__` with the inner `Schema` class.

apply() → BinaryIO
Apply the Strategy.

Returns The resulting file contents which would overwrite the target file.

4.2.1 Built-in Strategies

class scaraplate.strategies.**Overwrite**
Bases: *scaraplate.strategies.Strategy*

A simple strategy which always overwrites the target files with the ones from the template.

class Schema
An empty default schema which doesn't accept any parameters.

class scaraplate.strategies.**IfMissing**
Bases: *scaraplate.strategies.Strategy*

A strategy which writes the file from the template only if it doesn't exist in target.

class Schema
An empty default schema which doesn't accept any parameters.

class scaraplate.strategies.**SortedUniqueLines**
Bases: *scaraplate.strategies.Strategy*

A strategy which combines both template and target files, sorts the combined lines and keeps only unique ones.

However, the comments in the beginning of the files are treated differently. They would be stripped from the target and replaced with the ones from the template. The most common usecase for this are the License headers.

Sample `scaraplate.yaml` excerpt:

```
strategies_mapping:
  MANIFEST.in:
    strategy: scaraplate.strategies.SortedUniqueLines
  .gitignore:
    strategy: scaraplate.strategies.SortedUniqueLines
```

class Schema
Allowed params:

- `comment_pattern` [`^ *([;#%]|//)`] – a PCRE pattern which should match the line with a comment.

class scaraplate.strategies.**TemplateHash**
Bases: *scaraplate.strategies.Strategy*

A strategy which appends to the target file a git commit hash of the template being applied; and the subsequent applications of the same template for this file are ignored until the HEAD commit of the template changes.

This strategy is useful when a file needs to be different from the template but there's no suitable automated strategy yet, so it should be manually resynced on template updates.

This strategy overwrites the target file on each new commit in the template. There's also a *RenderedTemplateFileHash* strategy which does it less frequently: only when the source file from the template has changes.

Sample `scaraplate.yaml` excerpt:

```
strategies_mapping:
  setup.py:
    strategy: scaraplate.strategies.TemplateHash
    config:
      line_comment_start: '#'
      max_line_length: 87
      max_line_linter_ignore_mark: '  # noqa'
  Jenkinsfile:
    strategy: scaraplate.strategies.TemplateHash
    config:
      line_comment_start: '//'
```

This would result in the following:

setup.py:

```
...file contents...

# Generated by https://github.com/rambler-digital-solutions/scaraplate
# From https://github.com/rambler-digital-solutions/scaraplate-example-template/
↪commit/1111111111111111111111111111111111111111  # noqa
```

Jenkinsfile:

```
...file contents...

// Generated by https://github.com/rambler-digital-solutions/scaraplate
// From https://github.com/rambler-digital-solutions/scaraplate-example-template/
↪commit/1111111111111111111111111111111111111111
```

class Schema

Allowed params:

- `line_comment_start` [`#`] – The prefix which should be used to start a line comment.
- `max_line_length` [*None*] – The maximum line length for the appended line comments after which the `max_line_linter_ignore_mark` suffix should be added.
- `max_line_linter_ignore_mark` [`# noqa`] – The linter's line ignore mark for the appended line comments which are longer than `max_line_length` columns. The default `# noqa` mark silences `flake8`.

class scaraplate.strategies.**RenderedTemplateFileHash**

Bases: *scaraplate.strategies.TemplateHash*

A strategy which appends to the target file a hash of the rendered template file; and the subsequent applications of the same template for this file are ignored until the rendered file has changes.

This strategy is similar to *TemplateHash* with the difference that the target file is rewritten less frequently: only when the hash of the source file from the template is changed.

New in version 0.2.

Sample `scaraplate.yaml` excerpt:

```
strategies_mapping:
    setup.py:
      strategy: scaraplate.strategies.RenderedTemplateFileHash
      config:
        line_comment_start: '#'
        max_line_length: 87
        max_line_linter_ignore_mark: '  # noqa'
    Jenkinsfile:
      strategy: scaraplate.strategies.RenderedTemplateFileHash
      config:
        line_comment_start: '//'
```

This would result in the following:

setup.py:

```
...file contents...

# Generated by https://github.com/rambler-digital-solutions/scaraplate
# RenderedTemplateFileHash d2671228e3dfc3e663bfaf9b5b151ce8
# From https://github.com/rambler-digital-solutions/scaraplate-example-template/
↪commit/1111111111111111111111111111111111111111  # noqa
```

Jenkinsfile:

```
...file contents...

// Generated by https://github.com/rambler-digital-solutions/scaraplate
// RenderedTemplateFileHash d2as1228eb7233e663bfaf9b5b151ce8
// From https://github.com/rambler-digital-solutions/scaraplate-example-template/
↪commit/1111111111111111111111111111111111111111
```

class Schema

Allowed params:

- `line_comment_start` [`#`] – The prefix which should be used to start a line comment.
- `max_line_length` [*None*] – The maximum line length for the appended line comments after which the `max_line_linter_ignore_mark` suffix should be added.
- `max_line_linter_ignore_mark` [`# noqa`] – The linter's line ignore mark for the appended line comments which are longer than `max_line_length` columns. The default `# noqa` mark silences `flake8`.

class scaraplate.strategies.**ConfigParserMerge**

Bases: *scaraplate.strategies.Strategy*

A strategy which merges INI-like files (with `configparser`).

The resulting file is the one from the template with the following modifications:

- Comments are stripped
- INI file is reformatted (whitespaces are cleaned, sections and keys are sorted)

- Sections specified in the `preserve_sections` config list are preserved from the target file.
- Keys specified in the `preserve_keys` config list are preserved from the target file.

This strategy cannot be used to merge config files which contain keys without a preceding section declaration (e.g. `.editorconfig` won't work).

Sample `scaraplate.yaml` excerpt:

```
strategies_mapping:
  .pylintrc:
    strategy: scaraplate.strategies.ConfigParserMerge
    config:
      preserve_sections: []
      preserve_keys:
      - sections: ^MASTER$
        keys: ^extension-pkg-whitelist$
      - sections: ^TYPECHECK$
        keys: ^ignored-

  tox.ini:
    strategy: scaraplate.strategies.ConfigParserMerge
    config:
      preserve_sections:
      - sections: ^tox$
      preserve_keys:
      - sections: ^testenv
        keys: ^extras$

  pytest.ini:
    strategy: scaraplate.strategies.ConfigParserMerge
    config:
      preserve_sections: []
      preserve_keys:
      - sections: ^pytest$
        keys: ^python_files$

  .isort.cfg:
    strategy: scaraplate.strategies.ConfigParserMerge
    config:
      preserve_sections: []
      preserve_keys:
      - sections: ^settings$
        keys: ^known_third_party$
```

class Schema

Allowed params:

- `preserve_keys` (required) – the list of config keys which should be preserved from the target file. Values schema:
  - `sections` (required) – a PCRE pattern matching sections containing the keys to preserve.
  - `keys` (required) – a PCRE pattern matching keys in the matched sections.
- `preserve_sections` (required) – the list of config sections which should be preserved from the target file. If the matching section exists in the template, it would be fully overwritten. Values schema:
  - `sections` (required) – a PCRE pattern matching sections which should be preserved from the target.

  - `excluded_keys` [*None*] – a PCRE pattern matching the keys which should not be overwritten in the template when preserving the section.

class scaraplate.strategies.**SetupCfgMerge**

Bases: *scaraplate.strategies.ConfigParserMerge*

A strategy which merges the Python's `setup.cfg` file.

Based on the *ConfigParserMerge* strategy, additionally containing a `merge_requirements` config option for merging the lists of Python requirements between the files.

Sample `scaraplate.yaml` excerpt:

```
strategies_mapping:
  setup.cfg:
    strategy: scaraplate.strategies.SetupCfgMerge
    config:
      merge_requirements:
      - sections: ^options$
        keys: ^install_requires$
      - sections: ^options\.extras_require$
        keys: ^develop$
      preserve_keys:
      - sections: ^tool:pytest$
        keys: ^testpaths$
      - sections: ^build$
        keys: ^executable$
      preserve_sections:
      - sections: ^mypy-
      - sections: ^options\.data_files$
      - sections: ^options\.entry_points$
      - sections: ^options\.extras_require$
```

class Schema

Allowed params:

- `preserve_keys` (required) – the list of config keys which should be preserved from the target file. Values schema:
  - `sections` (required) – a PCRE pattern matching sections containing the keys to preserve.
  - `keys` (required) – a PCRE pattern matching keys in the matched sections.
- `preserve_sections` (required) – the list of config sections which should be preserved from the target file. If the matching section exists in the template, it would be fully overwritten. Values schema:
  - `sections` (required) – a PCRE pattern matching sections which should be preserved from the target.
  - `excluded_keys` [*None*] – a PCRE pattern matching the keys which should not be overwritten in the template when preserving the section.
- `merge_requirements` (required) – the list of config keys containing the lists of Python requirements which should be merged together. Values schema:
  - `sections` (required) – a PCRE pattern matching sections containing the keys with requirements.
  - `keys` (required) – a PCRE pattern matching keys in the matched sections.

4.3 Template Git Remotes

Scaraplate assumes that the template dir is a git repo.

Strategies receive a *scaraplate.template.TemplateMeta* instance which contains URLs to the template's project and the HEAD git commit on a git remote's web interface (such as GitHub). These URLs might be rendered in the target files by the strategies.

Scaraplate has built-in support for some popular git remotes. The remote is attempted to be detected automatically, but if that fails, it should be specified manually.

Sample `scaraplate.yaml` excerpt:

```
git_remote_type: scaraplate.gitremotes.GitHub
```

class scaraplate.template.**TemplateMeta**
Bases: tuple

Metadata of the template's git repo status.

commit_hash
Alias for field number 1

commit_url
Alias for field number 2

git_project_url
Alias for field number 0

head_ref
Alias for field number 4

is_git_dirty
Alias for field number 3

class scaraplate.gitremotes.**GitRemote**(*remote: str*)
Bases: abc.ABC

Base class for a git remote implementation, which generates http URLs from a git remote (either ssh of http) and a commit hash.

__init__(*remote: str*) → None
Init the git remote.

Parameters **remote** – A git remote, either ssh or http(s).

commit_url(*commit_hash: str*) → str
Return a commit URL at the given git remote.

Parameters **commit_hash** – Git commit hash.

project_url() → str
Return a project URL at the given git remote.

4.3.1 Built-in Git Remotes

class scaraplate.gitremotes.**GitLab**(*remote: str*)
Bases: *scaraplate.gitremotes.GitRemote*

GitLab git remote implementation.

class scaraplate.gitremotes.**GitHub**(*remote: str*)
Bases: *scaraplate.gitremotes.GitRemote*

GitHub git remote implementation.

class scaraplate.gitremotes.**BitBucket**(*remote: str*)
Bases: *scaraplate.gitremotes.GitRemote*

BitBucket git remote implementation.

4.4 Rollup Automation

Once you get comfortable with manual rollups, you might want to set up regularly executed automated rollups.

At this moment scaraplate doesn't provide a CLI for that, but there's a quite extensible Python code which simplifies implementation of custom scenarios.

4.4.1 Supported automation scenarios

GitLab Merge Request

GitLab integration requires python-gitlab package, which can be installed with:

```
pip install 'scaraplate[gitlab]'
```

Sample `rollup.py` script:

```
from scaraplate import automatic_rollup
from scaraplate.automation.gitlab import (
    GitLabCloneTemplateVCS,
    GitLabMRProjectVCS,
)


automatic_rollup(
    template_vcs_ctx=GitLabCloneTemplateVCS.clone(
        project_url="https://mygitlab.example.org/myorg/mytemplate",
        private_token="your_access_token",
        clone_ref="master",
    ),
    project_vcs_ctx=GitLabMRProjectVCS.clone(
        gitlab_url="https://mygitlab.example.org",
        full_project_name="myorg/mytargetproject",
        private_token="your_access_token",
        changes_branch="scheduled-template-update",
        clone_ref="master",
    ),
)
```

This script would do the following:

1. `git clone` the template repo to a tempdir;
2. `git clone` the project repo to a tempdir;
3. Run `scaraplate rollup ... --no-input`;

4. Would do nothing if rollup didn't change anything; otherwise it would create a commit with the changes, push it to the `scheduled-template-update` branch and open a GitLab Merge Request from this branch.

If a MR already exists, *GitLabMRProjectVCS* does the following:

1. A one-commit git diff is compared between the already existing MR's branch and the locally committed branch (in a tempdir). If diffs are equal, nothing is done.
2. If diffs are different, the existing MR's branch is removed from the remote, effectively closing the old MR, and a new branch is pushed, which is followed by creation of a new MR.

To have this script run daily, crontab can be used. Assuming that the script is located at `/opt/rollup.py` and the desired time for execution is 9:00, it might look like this:

```
$ crontab -e
# Add the following line:
00 9 * * *  python3 /opt/rollup.py
```

Git push

GitLabCloneTemplateVCS and *GitLabMRProjectVCS* are based off *GitCloneTemplateVCS* and *GitCloneProjectVCS* correspondingly. GitLab classes add GitLab-specific git-clone URL generation and Merge Request creation. The rest (git clone, commit, push) is done in the *GitCloneTemplateVCS* and *GitCloneProjectVCS* classes.

GitCloneTemplateVCS and *GitCloneProjectVCS* classes work with any git remote. If you're okay with just pushing a branch with updates (without opening a Merge Request/Pull Request), then you can use the following:

Sample `rollup.py` script:

```
from scaraplate import automatic_rollup, GitCloneProjectVCS, GitCloneTemplateVCS


automatic_rollup(
    template_vcs_ctx=GitCloneTemplateVCS.clone(
        clone_url="https://github.com/rambler-digital-solutions/scaraplate-example-
↪template.git",
        clone_ref="master",
    ),
    project_vcs_ctx=GitCloneProjectVCS.clone(
        clone_url="https://mygit.example.org/myrepo.git",
        clone_ref="master",
        changes_branch="scheduled-template-update",
        commit_author="scaraplate <yourorg@yourcompany>",
    ),
)
```

4.4.2 Python API

scaraplate.automation.base.**automatic_rollup**(*, *template_vcs_ctx: AbstractContextManager[TemplateVCS]*, *project_vcs_ctx: AbstractContextManager[ProjectVCS]*, *extra_context: Optional[Mapping[str, str]] = None*) → None

The main function of the automated rollup implementation.

This function accepts two context managers, which should return two classes: *TemplateVCS* and *ProjectVCS*, which represent the cloned template and target project correspondingly.

The context managers should prepare the repos, e.g. they should create a temporary directory, clone a repo there, and produce a *TemplateVCS* or *ProjectVCS* class instance.

This function then applies scaraplate rollup of the template to the target project in *no-input mode*. If the target project contains any changes (as reported by *ProjectVCS.is_dirty()*), they will be committed by calling *ProjectVCS.commit_changes()*.

New in version 0.2.

class scaraplate.automation.base.**TemplateVCS**
Bases: abc.ABC

A base class representing a template retrieved from a VCS (probably residing in a temporary directory).

The resulting directory with template must be within a git repository, see *Scaraplate Template* for details. But it doesn't mean that it must be retrieved from git. Template might be retrieved from anywhere, it just has to be in git at the end. That git repo will be used to fill the *TemplateMeta* structure.

dest_path
Path to the root directory of the template.

template_meta
TemplateMeta filled using the template's git repo.

class scaraplate.automation.base.**ProjectVCS**
Bases: abc.ABC

A base class representing a project retrieved from a VCS (probably residing in a temporary directory).

The project might use any VCS, at this point there're no assumptions made by scaraplate about the VCS.

commit_changes(*template_meta: scaraplate.template.TemplateMeta*) → None
Commit the changes made to the project. This method is responsible for delivering the changes back to the place the project was retrieved from. For example, if the project is using `git` and it was cloned to a temporary directory, then this method should commit the changes and push them back to git remote.

This method will be called only if *ProjectVCS.is_dirty()* has returned True.

dest_path
Path to the root directory of the project.

is_dirty() → bool
Tell whether the project has any changes not committed to the VCS.

class scaraplate.automation.git.**GitCloneTemplateVCS**(*template_path: pathlib.Path*, *template_meta: scaraplate.template.TemplateMeta*)
Bases: *scaraplate.automation.base.TemplateVCS*

A ready to use *TemplateVCS* implementation which:

- Uses git
- Clones a git repo with the template to a temporary directory (which is cleaned up afterwards)
- Allows to specify an inner dir inside the git repo as the template root (which is useful for monorepos)

classmethod clone(*clone_url: str*, ***, *clone_ref: Optional[str] = None*, *monorepo_inner_path: Optional[pathlib.Path] = None*) → Iterator[scaraplate.automation.git.GitCloneTemplateVCS]
Provides an instance of this class by issuing `git clone` to a tempdir when entering the context manager. Returns a context manager object which after `__enter__` returns an instance of this class.

Parameters

- **clone_url** – Any valid `git clone` url.
- **clone_ref** – Git ref to checkout after clone (i.e. branch or tag name).
- **monorepo_inner_path** – Path to the root dir of template relative to the root of the repo. If `None`, the root of the repo will be used as the root of template.

class scaraplate.automation.git.**GitCloneProjectVCS**(*project_path: pathlib.Path*, *git: scaraplate.automation.git.Git*, ***, *changes_branch: str*, *commit_author: str*, *commit_message_template: str*)

Bases: *scaraplate.automation.base.ProjectVCS*

A ready to use *ProjectVCS* implementation which:

- Uses git
- Clones a git repo with the project to a temporary directory (which is cleaned up afterwards)
- Allows to specify an inner dir inside the git repo as the project root (which is useful for monorepos)
- Implements *ProjectVCS.commit_changes()* as `git commit` + `git push`.

classmethod clone(*clone_url: str*, ***, *clone_ref: Optional[str] = None*, *monorepo_inner_path: Optional[pathlib.Path] = None*, *changes_branch: str*, *commit_author: str*, *commit_message_template: str = 'Scheduled template update ({update_time:%Y-%m-%d})\n\n* scaraplate version: {scaraplate_version}\n* template commit: {template_meta.commit_url}\n* template ref: {template_meta.head_ref}\n'*)
→ Iterator[scaraplate.automation.git.GitCloneProjectVCS]

Provides an instance of this class by issuing `git clone` to a tempdir when entering the context manager. Returns a context manager object which after `__enter__` returns an instance of this class.

Parameters

- **clone_url** – Any valid `git clone` url.
- **clone_ref** – Git ref to checkout after clone (i.e. branch or tag name).
- **monorepo_inner_path** – Path to the root dir of project relative to the root of the repo. If `None`, the root of the repo will be used as the root of project.
- **changes_branch** – The branch name where the changes should be pushed in the remote. Might be the same as `clone_ref`. Note that this branch is never force-pushed. If upon push the branch already exists in remote and its one-commit diff is different from the one-commit diff of the just created local branch, then the remote branch will be deleted and the local branch will be pushed to replace the previous one.
- **commit_author** – Author name to use for `git commit`, e.g. `John Doe <john@example.org>`.
- **commit_message_template** – `str.format()` template which is used to produce a commit message when committing the changes. Available format variables are:
  - `update_time` [`datetime.datetime`] – the time of update
  - `scaraplate_version` [`str`] – scaraplate package version
  - `template_meta` [*TemplateMeta*] – template meta returned by `TemplateVCS.template_meta()`

class scaraplate.automation.gitlab.**GitLabCloneTemplateVCS**(*git_clone: scaraplate.automation.git.GitCloneTemplateVCS*)

Bases: *scaraplate.automation.base.TemplateVCS*

A class which extends *GitCloneTemplateVCS* with GitLab-specific `clone_url` generation.

classmethod clone(*project_url: str*, *private_token: Optional[str] = None*, ***, *clone_ref: Optional[str] = None*, *monorepo_inner_path: Optional[pathlib.Path] = None*) → Iterator[scaraplate.automation.gitlab.GitLabCloneTemplateVCS]

Same as *GitCloneTemplateVCS.clone()* except that `clone_url` is replaced with `project_url` and `private_token`.

The `private_token` allows to clone private repos, which are visible only for an authenticated user.

Parameters

- **project_url** – A URL to a GitLab project, e.g. `https://gitlab.example.org/myorganization/myproject`.
- **private_token** – GitLab access token, see https://docs.gitlab.com/ce/api/#oauth2-tokens.

class scaraplate.automation.gitlab.**GitLabMRProjectVCS**(*git_clone: scaraplate.automation.git.GitCloneProjectVCS*, ***, *gitlab_project*, *mr_title_template: str*, *mr_description_markdown_template: str*)

Bases: *scaraplate.automation.base.ProjectVCS*

A class which extends *GitCloneProjectVCS* with GitLab-specific `clone_url` generation and opens a GitLab Merge Request after `git push`.

classmethod clone(*gitlab_url: str*, *full_project_name: str*, *private_token: str*, ***, *mr_title_template: str = 'Scheduled template update ({update_time:%Y-%m-%d})'*, *mr_description_markdown_template: str = '* scaraplate version: `{scaraplate_version}`\n* template commit: {template_meta.commit_url}\n* template ref: {template_meta.head_ref}\n'*, *commit_author: Optional[str] = None*, ***kwargs*) → Iterator[scaraplate.automation.gitlab.GitLabMRProjectVCS]

Same as *GitCloneProjectVCS.clone()* with the following exceptions:

- `clone_url` is replaced with `gitlab_url`, `full_project_name` and `private_token`.
- A GitLab Merge Request (MR) is opened after a successful `git push`.

The `private_token` allows to clone private repos, which are visible only for an authenticated user.

As in *GitCloneProjectVCS.clone()*, the `changes_branch` might be the same as `clone_ref`. In this case no MR will be opened.

A MR will be created only if there're any changes produced by scaraplate rollup. If a `changes_branch` is already present in remote (i.e. there is a previous automatic rollup which wasn't merged yet), there're two possibilities:

- If one-commit diffs between the remote's `changes_branch` and the local `changes_branch` are the same, nothing is done. It means that a MR already exists and it has the same patch as the one which was just produced locally.
- If the diffs are different, the remote branch will be deleted, effectively closing the old MR, and a new one will be pushed instead, and a new MR will be opened.

The opened MRs are expected to be merged manually.

Parameters

- **gitlab_url** – A URL to the GitLab instance, e.g. `https://gitlab.example.org`.
- **full_project_name** – Project name within gitlab, e.g. `myorganization/myproject`.
- **private_token** – GitLab access token, see https://docs.gitlab.com/ce/api/#oauth2-tokens.
- **mr_title_template** – `str.format()` template which is used to produce a MR title. Available format variables are:
  - `update_time` [`datetime.datetime`] – the time of update
  - `template_meta` [*TemplateMeta*] – template meta returned by `TemplateVCS.template_meta()`
- **mr_description_markdown_template** – `str.format()` template which is used to produce a MR description (which will be rendered as markdown). Available format variables are:
  - `update_time` [`datetime.datetime`] – the time of update
  - `scaraplate_version` [`str`] – scaraplate package version
  - `template_meta` [*TemplateMeta*] – template meta returned by `TemplateVCS.template_meta()`
- **commit_author** – Author name to use for `git commit`, e.g. `John Doe <john@example.org>`. If `None`, will be retrieved from GitLab as the name of the currently authenticated user (using `private_token`).

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols

A

B

C

D

G

H

I

O

P

R

S

T