diecutter Documentation

Release 0.8.dev0

Remy Hubscher

May 04, 2016

Contents

diecutter is a web application around file generation:

- templates are the resources ;
- the most common operation is to **POST data to templates in order to retrieve generated files**.

diecutter can render single files and directories. Directories are rendered as archives.

CHAPTER 1

Example

`GET` raw content of a template:

```
$ curl -X GET http://diecutter.io/api/greetings.txt
{{ greetings|default('Hello') }} {{ name }}!
```

`POST` data to the template and retrieve generated content:

```
$ curl -X POST -d name=world http://diecutter.io/api/greetings.txt
Hello world!
```

CHAPTER 2

Project status

Although under active development, *diecutter* already works, so give it a try!.

Check milestones and vision for details about the future.

Also notice that *diecutter* is part of an ecosystem:

- piecutter is the core Python API. It provides stuff like template engines or template loaders.
- *diecutter* implements a WSGI application and REST interface on top of *piecutter*.
- diecutter-index is a proof-of-concept project for an online template registry.
- http://diecutter.io is the SAAS platform running *diecutter* ecosystem.

See also alternatives and related projects section in documentation.

CHAPTER 3

Resources

- Online demo: http://diecutter.io
- Documentation: http://diecutter.readthedocs.io
- PyPI page: http://pypi.python.org/pypi/diecutter
- Bugtracker: https://github.com/diecutter/diecutter/issues
- Changelog: https://diecutter.readthedocs.io/en/latest/about/changelog.html
- Roadmap: https://github.com/diecutter/diecutter/issues/milestones
- Code repository: https://github.com/diecutter/diecutter
- Continuous integration: https://travis-ci.org/diecutter/diecutter
- IRC Channel: irc://irc.freenode.net/#diecutter

CHAPTER 4

Contents

4.1 Client howto

This section focus on usage of *diecutter* service API, i.e. usage of *diecutter* from client's point of view.

4.1.1 General API information

Here are features of *diecutter* API's root endpoint, i.e. `/`.

Note: In the examples below, let's communicate with a *diecutter* server using Python requests.

This *diecutter* serves diecutter's demo templates.

The `diecutter_url` variable holds root URL of *diecutter* service, i.e. something like http://diecutter.io/api/.

API version

diecutter root API endpoint returns an "hello" message and shows software version:

```
>>> response = requests.get(diecutter_url)
>>> response.status_code
200
>>> print response.json()['diecutter']
Hello
>>> import diecutter
>>> response.json()['version'] == diecutter.__version__
True
```

List supported engines

Ask the *diecutter* server API index to get the list of engines:

```
>>> response = requests.get(diecutter_url)
>>> response.json()['engines']
[u'django', u'filename', u'jinja2']
```

4.1.2 Files

This document explains how to render single files using *diecutter* web API. See also Directories about how to render multiple files at once.

Here is service's API overview:

- *GET*: retrieve information about template, typically raw template content.
- *POST*: render single template file against data.
- *PUT*: upload template file from client to server.
- other HTTP verbs aren't implemented yet.

Note: In the examples below, let's communicate with a *diecutter* server using Python requests.

This *diecutter* serves diecutter's demo templates.

The `diecutter_url` variable holds root URL of *diecutter* service, i.e. something like http://diecutter.io/api/.

GET

Let's work on file resource greetings.txt.

A `GET` request returns raw template content:

```
>>> response = requests.get(diecutter_url + 'greetings.txt')
>>> response.status_code
200
>>> print response.content
{{ greetings|default('Hello') }} {{ name }}!
```

Of course, if template does not exist, it returns a 404:

```
>>> response = requests.get(diecutter_url + 'i-do-not-exist.txt')
>>> response.status_code
404
```

POST

`POST` data to template resource in order to retrieve generated file:

```
>>> greetings_url = diecutter_url + 'greetings.txt'
>>> response = requests.post(greetings_url, data={'name': 'world'})
>>> response.status_code
200
>>> print response.content
Hello world!

>>> response = requests.post(greetings_url, {'greetings': u'Greetings',
...                                          'name': u'professor Falken'})
>>> print response.content
Greetings professor Falken!
```

See also Targeting template engines about using specific template engines.

PUT

Use `PUT` to upload template from client to server:

```
>>> name = 'hello'
>>> server_side_name = os.path.join(diecutter_template_dir, name)
>>> os.path.exists(server_side_name)
False
>>> url = diecutter_url + name
>>> files = {'file': ('client-side-name', 'Hello {{ who }}\n')}
>>> response = requests.put(url, files=files)
>>> response.status_code
201
>>> os.path.exists(server_side_name)
True
>>> print open(server_side_name).read()
Hello {{ who }}
```

Subdirectories are created on the fly:

```
>>> name = 'some/nested/directories/hello'
>>> server_side_name = os.path.join(diecutter_template_dir, name)
>>> os.path.exists(server_side_name)
False
>>> url = diecutter_url + name
>>> files = {'file': ('client-side-name', 'Hello {{ who }}\n')}
>>> response = requests.put(url, files=files)
>>> response.status_code
201
>>> os.path.exists(server_side_name)
True
>>> print open(server_side_name).read()
Hello {{ who }}
```

Warning: Sometimes, you don't want users to be able to PUT files on your server. That's why diecutter service can be configured as "read only". In that case, it is up to you to manage files that live in diecutter's template directory (which is just a directory in the filesystem).
As an example, diecutter's online demo is readonly, and templates are synchronized with source code on project's repository.

4.1.3 Directories

This document explains how to render directories (i.e. multiple files) using *diecutter* web API. See also Files about how to render single files.

Here is service's API overview:

- *GET*: list files in directory.
- *POST*: render templates in directory against context, as an archive.
- *PUT*: directories are automatically created when you PUT a file in it.
- other HTTP verbs aren't implemented yet.

Note: In the examples below, let's communicate with a *diecutter* server using Python requests.

This *diecutter* serves diecutter's demo templates.

The `diecutter_url` variable holds root URL of *diecutter* service, i.e. something like http://diecutter.io/api/.

PUT

In order to create a directory, `PUT` a file in it.

As an example, let's create some "greetings" directory with "hello.txt" and "goodbye.txt" inside:

```
>>> name = 'greetings/hello.txt'
>>> files = {'file': ('client-side-name', 'Hello {{ name }}\n')}
>>> response = requests.put(diecutter_url + name, files=files)
>>> response.status_code
201
>>> name = 'greetings/goodbye.txt'
>>> files = {'file': ('client-side-name', 'Goodbye {{ name }}\n')}
>>> response = requests.put(diecutter_url + name, files=files)
>>> response.status_code
201
```

There is currently no way to create an empty directory. And no way to upload multiple files at once.

GET

`GET` lists files in directory, recursively:

```
>>> response = requests.get(diecutter_url + 'greetings/')
>>> response.status_code
200
>>> print response.content
goodbye.txt
hello.txt
```

Notice that, if you don't set the trailing slash, you get the same list with folder name as prefix:

```
>>> response = requests.get(diecutter_url + 'greetings')
>>> response.status_code
200
>>> print response.content
greetings/goodbye.txt
greetings/hello.txt
```

POST

`POST` renders directory against data, and the result is an archive.

```
>>> response = requests.post(
...     diecutter_url + 'greetings/',
...     data={'name': 'Remy'})
>>> response.status_code
200
>>> archive = tarfile.open(fileobj=StringIO(response.content))
>>> print archive.getnames()
['goodbye.txt', 'hello.txt']
>>> print archive.extractfile('hello.txt').read()
Hello Remy
```

```
>>> print archive.extractfile('goodbye.txt').read()
Goodbye Remy
```

By default, archives are in tar.gz format. See *accept header* below for alternatives.

Rendering only parts of a directory

You can render only parts of a directory! It means you can render sub-directories or single files:

```
>>> response = requests.post(
...     diecutter_url + 'greetings/hello.txt',
...     data={'name': 'world'})
>>> print response.content
Hello world
```

This is because *diecutter* handles templates just as files and directories, in any filesystem it can read. If it can read a directory or a file, they can be used as a template, and so can be the sub-directories and files. *diecutter* does not need you to register templates, it only needs to be able to read the "filesystem" (repositories are kind of filesystems too) holding templates.

Dynamic trees

By default, all files in directory are rendered, and the context data does not vary. See Dynamic directory trees templates if you need to dynamically alter the list of files or the context data.

Trailing slash

Like with *GET*, the trailing slash affects filenames: without trailing slash, filenames are prefixed with directory name.

```
>>> response = requests.post(
...     diecutter_url + 'greetings',
...     data={'name': 'Remy'})
>>> archive = tarfile.open(fileobj=StringIO(response.content))
>>> print archive.getnames()
['greetings/goodbye.txt', 'greetings/hello.txt']
```

In the archive, dates reflect generation time:

```
>>> time_floor = int(time.time())  # Time before generation.
>>> response = requests.post(
...     diecutter_url + 'greetings/',
...     data={'name': 'Remy'})
>>> archive = tarfile.open(fileobj=StringIO(response.content))
>>> time_ceil = time.time()  # Time after generation.
>>> info = archive.getmember('hello.txt')
>>> info.mtime != 0
True
>>> info.mtime >= time_floor
True
>>> info.mtime <= time_ceil
True
```

accept header

By default, *diecutter* returns TAR.GZ archives. You can get the content as a ZIP archive using the HTTP `accept` header:

```
>>> response = requests.post(
...     diecutter_url + 'greetings/',
...     data={'name': 'Remy'},
...     headers={'accept': 'application/zip'})
>>> import zipfile
>>> zip_filename = os.path.join(diecutter_template_dir, 'response.zip')
>>> open(zip_filename, 'w').write(response.content)
>>> zipfile.is_zipfile(zip_filename)  # File is actually a ZIP.
True
>>> archive = zipfile.ZipFile(zip_filename)
>>> archive.testzip() is None  # ZIP integrity is OK.
True
>>> print archive.namelist()
['goodbye.txt', 'hello.txt']
>>> print archive.read('hello.txt')
Hello Remy
```

Supported archive formats

You can see all supported "accept" headers by requesting an unknown mime type:

```
$ curl -X POST --header "accept:fake/mime-type" -d name="world" http://localhost:8106/greetings
406 Not Acceptable

The server could not comply with the request since it is either malformed or
otherwise incorrect.


Supported mime types: */*, application/gzip, application/x-gzip,
application/zip
```

4.1.4 Posting input context data

When you perform POST requests on resources, you provide context data, i.e. variables and values.

Diecutter has builtin support for the following input content-types:

- application/x-www-form-urlencoded [1]: the default when you perform POST requests with `wget` or `curl` ;
- application/json [2]: JSON encoded data ;
- text/plain: INI-style plain text files [3].

Diecutter expects data to be provided as the body of the request. "multipart/form-data" requests aren't supported currently.

[1] http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.1
[2] http://json.org/
[3] https://en.wikipedia.org/wiki/INI_file

URL-encoded

```
# Default (implicit application/x-www-form-urlencoded content type).
curl -X POST -d 'who=world' http://localhost:8106/hello

# Explicit "application/x-www-form-urlencoded" content-type.
curl -X POST -d 'who=world' -H "Content-Type: application/x-www-form-urlencoded" http://localhost:810
```

JSON

```
curl -X POST -d '{"who": "world"}' -H "Content-Type: application/json" http://localhost:8106/hello
```

INI

```
# Flat.
curl -X POST -d 'who=world' -H "Content-Type: text/plain" http://localhost:8106/hello

# With sections.
cat > input.ini <<EOF
hello = world
[foo]
bar = baz
EOF
curl -X POST --data-binary '@input.ini' -H "Content-Type: text/plain" http://localhost:8106/foo
```

curl tips

- Pass content of a file using `@`.
- Pass content from standard input using `@-`.
- When posting *JSON* or *INI*, use `--data-binary`.

References

4.1.5 Targeting template engines

diecutter supports several template engines. This section explains how *diecutter* chooses the engine for a given template.

As a summary, *diecutter* follows the scenario below:

- `engine` *in query string*;
- *default engine*.

Notice that *diecutter* sets a *diecutter-engine header in responses*.

Note: In the examples below, let's communicate with a *diecutter* server using Python requests.

This *diecutter* serves diecutter's demo templates.

The `diecutter_url` variable holds root URL of *diecutter* service, i.e. something like http://diecutter.io/api/.

In query string

If the HTTP request has a known `engine` in query string, then *diecutter* uses it to render the template. Here is an example using `jinja2`:

```
>>> greetings_url = diecutter_url + 'greetings.txt?engine=jinja2'
>>> response = requests.post(greetings_url, {'name': 'world'})
>>> print response.text
Hello world!
>>> print response.headers['Diecutter-Engine']
jinja2
```

Here is an example using `django`:

```
>>> greetings_url = diecutter_url + 'greetings-django.txt?engine=django'
>>> response = requests.post(greetings_url, {'name': 'world'})
>>> print response.text
Hello world!

>>> print response.headers['Diecutter-Engine']
django
```

If you try to use an unknown template engine, then you get a 406:

```
>>> greetings_url = diecutter_url + 'greetings.txt?engine=unknown'
>>> response = requests.post(greetings_url, {'name': 'world'})
>>> response.status_code
406
```

Default engine

In case no specific engine was resolved for the current request, *diecutter* fallbacks to default engines. Default engines are setup in configuration:

- `diecutter.engine` is the default engine used to render files;
- `diecutter.filename_engine` is the default engine used to render filenames.

The configuration itself defaults to `jinja2`.

```
>>> greetings_url = diecutter_url + 'greetings.txt'
>>> response = requests.post(greetings_url, {'name': 'world'})
>>> print response.text
Hello world!
>>> print response.headers['Diecutter-Engine']
jinja2
```

diecutter-engine header in responses

diecutter sets the value of used engine as `Diecutter-Engine` header.

```
>>> greetings_url = diecutter_url + 'greetings.txt'
>>> response = requests.post(greetings_url, {'name': 'world'})
>>> print response.headers['Diecutter-Engine']
jinja2
```

Supported engines

Ask the *diecutter* server API index to get the list of engines:

```
>>> response = requests.get(diecutter_url)
>>> response.json()['engines']
[u'django', u'filename', u'jinja2']
```

Unsupported engines

diecutter returns a HTTP 406 error code in case an unsupported template engine was asked:

```
>>> greetings_url = diecutter_url + 'greetings.txt?engine=unknown'
>>> response = requests.post(greetings_url, {'name': 'world'})
>>> response.status_code
406
```

4.1.6 Variable-based output filenames

This document explains how to render dynamic filenames.

When rendering a directory, it is sometimes useful to use variables in the filenames to output.

Tip: This is useless for single files, since most clients allow you to choose the name of the output file.

Note: In the examples below, let's communicate with a *diecutter* server using Python requests.

This *diecutter* serves diecutter's demo templates.

The `diecutter_url` variable holds root URL of *diecutter* service, i.e. something like http://diecutter.io/api/.

When rendering directories, templates having `+variable+` in the filename will be resolved against the context data:

```
>>> url = diecutter_url + 'simple-tree/'
>>> response = requests.get(url)
>>> print response.content
+name+.txt
>>> response = requests.post(url, {'name': 'demo'})
>>> archive = tarfile.open(fileobj=StringIO(response.content))
>>> print archive.getnames()
['demo.txt']
```

Tip: **The "variable-based filenames" feature is meant for simple cases**. If you need advanced things for filenames, use the dynamic trees feature.

Notice that "variable-based filenames" behaviour is disabled when using the dynamic trees.

4.1.7 Dynamic directory trees templates

Directory trees can be computed from templates.

Thus you can use all features of template engines to render filenames, select templates or even alter context.

Note: In the examples below, let's communicate with a *diecutter* server using Python requests.

This *diecutter* serves diecutter's demo templates.

The `diecutter_url` variable holds root URL of *diecutter* service, i.e. something like http://diecutter.io/api/.

Use cases

While rendering a directory...

- skip some files based on variables ;
- render a single template several times with different output filenames ;
- alter template context data for some templates ;
- use loops, conditions... and all template-engine features...

The template tree template

When you POST to a directory, diecutter looks for special ".directory-tree" template in that directory. If present, it renders ".directory-tree" against context, decodes JSON, then iterates over items to actually render the directory.

Except when rendering a directory resource, ".directory-tree" template is a normal template resource file. Manage it as any other template file.

Example

Let's explain this feature with an example...

Let's work on directory resource `dynamic-tree/`.

```
>>> dynamic_tree_url = diecutter_url + 'dynamic-tree/'
```

It's a directory. So `GET` lists the templates it contains:

```
>>> response = requests.get(dynamic_tree_url)
>>> print response.content
.directory-tree
greeter.txt
```

`greeter.txt` is a template, with nothing special:

```
>>> response = requests.get(dynamic_tree_url + 'greeter.txt')
>>> print response.content
{{ greeter|default('Hello') }} {{ name|default('world') }}!
```

`directory-tree` is a also a template, with a special name:

```
>>> response = requests.get(dynamic_tree_url + '.directory-tree')
>>> print response.content
[
  {% for greeter in greeting_list|default(['hello', 'goodbye']) %}
    {
      "template": "greeter.txt",
```

```
      "filename": "{{ greeter }}.txt",
      "context": {"greeter": "{{ greeter }}"}
    }{% if not loop.last %},{% endif %}
  {% endfor %}
]
```

`directory-tree` template renders as a list of templates, in JSON:

```
>>> response = requests.post(dynamic_tree_url + '.directory-tree',
...                          {'greeting_list': [u'bonjour', u'bonsoir']})
>>> print response.content
[

    {
      "template": "greeter.txt",
      "filename": "bonjour.txt",
      "context": {"greeter": "bonjour"}
    },

    {
      "template": "greeter.txt",
      "filename": "bonsoir.txt",
      "context": {"greeter": "bonsoir"}
    }

]
```

JSON-encoded list items are dictionaries with the following keys:

- "template": relative path to a template, i.e. content to be rendered ;
- "filename": filename to return to the client ;
- "context": optional dictionary of context overrides.

When rendering a directory, *diecutter* first computes the tree of templates in the directory. By default, it just reads the filesystem. But if there is a `.directory-tree` file, then this special file is rendered first, and the result is used as the list of templates to render.

So, when you render the directory, you generate the files according to the generated `directory-tree` template:

```
>>> response = requests.post(dynamic_tree_url,
...                          {'name': u'Remy',
...                           'greeting_list': [u'bonjour', u'bonsoir']})
>>> archive = tarfile.open(fileobj=StringIO(response.content), mode='r:gz')
>>> print '\n'.join(archive.getnames())
bonjour.txt
bonsoir.txt
>>> print archive.extractfile('bonjour.txt').read()
bonjour Remy!
>>> print archive.extractfile('bonsoir.txt').read()
bonsoir Remy!
>>> archive.close()
```

Here, the `greeter.txt` template has been rendered several times, with different context data.

4.2 Server howto

This section explains how to setup, configure and run a diecutter server.

4.2.1 Install, configure, run

This project is open-source, published under BSD license. See License for details.

If you want to install a development environment, you should go to Contributing to diecutter documentation.

Prerequisites

- Python [1] version 2.7.

Install

Install "diecutter" package with your favorite tool. As an example with pip [2]:

```
pip install diecutter
```

Configure

Use diecutter's online demo [3] to generate your local diecutter configuration:

```
# Adapt "YOUR_TEMPLATE_DIR"!
wget -O diecutter.ini --post-data "template_dir=YOUR_TEMPLATE_DIR" http://diecutter.io/api/diecutter
```

diecutter.service

Python path to service class that implements *diecutter* API. Default is `diecutter.local.LocalService`.

Builtin services are:

- `diecutter.local.LocalService`
- `diecutter.github.GithubService`

diecutter.engine

Code of the default engine to use to render files. Default is `jinja2`.

diecutter.filename_engine

Code of the default engine to use to render filenames. Default is `filename`.

diecutter.engine.*

Mapping between engine codes and Python path to the template engine class. Defaults:

- django: `piecutter.engines.django:DjangoEngine`
- filename: `piecutter.engines.filename:FilenameEngine`

[1] http://python.org
[2] https://pypi.python.org/pypi/pip/
[3] http://diecutter.io/

- jinja2: `piecutter.engines.jinja:Jinja2Engine`

Note: *diecutter* itself does not implement engines. Engines are implemented as part of piecutter.

Run

`pserve` (paster's server) should have been installed automatically as part of diecutter's dependencies. Use it to run the service:

```
pserve diecutter.ini --reload
```

Check

Check it works:

```
curl http://localhost:8106
```

You should get an "hello" with diecutter's version.

References

4.3 Python API

diecutter is written in Python [1]. Here are some tips about *diecutter*'s internals:

- *diecutter* is built on top of a third-party project called piecutter [2]. *piecutter* provides things such as template engines and template loaders. In most cases, i.e. except you explicitely need *diecutter*'s WSGI layer, have a look at *piecutter*'s Python API.
- *diecutter* provides a WSGI layer and a REST API on top of *piecutter*.

Note: At the moment, authors put efforts in *piecutter*'s API, because it is more general purpose than *diecutter*'s. That said, if you have an use case for *diecutter* API, let us know! Feedback, reviews and contributions are welcome ;)

Notes & references

4.4 About diecutter

This section is about the diecutter project itself.

4.4.1 Vision

Diecutter is about file generation. Its primary goal is to provide an easy way to render templates against data.

Some leitmotivs:

[1] http://python.org
[2] https://pypi.python.org/pypi/piecutter

- focus on file generation.
- in most use cases, don't bother users with software installation and configuration.
- when users have specific needs, they can easily setup a custom service.

API

API design drives diecutter's development.

Server software

Diecutter provides a default implementation of the service, built with Python.

Framework

Diecutter is a framework. It must provide material that makes it easy to connect to other tools and easy to extend.

SAAS platform

Diecutter is developed with SAAS in mind. The online demo is a draft.

4.4.2 Alternatives and related projects

This document presents other projects that provide similar or complementary functionalities. It focuses on differences with *diecutter*.

Note: This document reflects *diecutter*'s authors point of view, which is, inherently, a partial one. Contact authors if you feel something is wrong.

---

diecutter ecosystem

diecutter is part of an ecosystem:

- piecutter [1] is the core Python API. It provides stuff like template engines or template loaders.
- *diecutter* implements a WSGI application and REST interface on top of *piecutter*.
- diecutter-index is a proof-of-concept project for an online template registry.
- http://diecutter.io is the SAAS platform running *diecutter* ecosystem.
- diecutter organization on Github has been created to manage every related projects in a single place.

[1] https://pypi.python.org/pypi/piecutter/

paster create

PasteScript [2] provides a `paster create` command to generate files from templates.

diecutter's authors tried PasteScript, and found it too complicated for most use cases. They felt that the more they generated files with *PasteScript*, the more they spent time on *PasteScript*'s specific stuff. With *diecutter*, they tried to focus on using templates to generate files.

Let's highlight some differences between *PasteScript* and *diecutter*:

PasteScript	diecutter
`paster create` is a command. Runs on local machine. You need PasteScript installed on every computer (or project) where you use templates.	Diecutter is a web service. You can use a shared diecutter server, but it is also easy to deploy on a local machine.
You have to register templates: • create template class (Cheetah's `Template`), • update template package's entry points (`setup.py`), • update your environment (`setup.py install`).	You PUT files just like in a filesystem.
You distribute templates in Python packages.	You carry templates (folders and files) as you like. No packaging.
Can collect input data via interactive shell prompts.	No builtin client (that's part of the Vision). Listing variables isn't implemented yet (that's part of the plans).
You declare sets of templates. You can't render only some part of it, you have to render the whole set.	You can render a directory as a set, but you can also render any single part of it, individually.
Templates are Python classes. That allows developers to implement various tasks in hooks.	Uses only templates. Even to dynamically generate folders.
Asks you whether to ovewrite files in output directory, and even shows you the diff!	That's not a job for diecutter. You can do that (better) with Git, Mercurial and many other tools.
Uses Cheetah [3] template engine.	Supports several template engines, such as Jinja [4] and Django [5]. Cheetah [3] is in the plans.
Not only about generating files. PasteScript is a big project that does several things.	Focuses on generating files.

collective.generic.webbuilder

collective.generic.webbuilder [6] is a file generation service, built on top of *paster create*. It is a web interface to `paster create`.

Thus, some differences are directly related to *paster create*, such as you define sets of templates, or have to register templates (collective.generic.webbuilder uses ZCML [7]).

Collective.generic.webbuilder inspired diecutter.

Diecutter's authors felt that collective.generic.webbuilder is a tool made by paster users, for paster users. Mostly developers. With diecutter, they tried to make things easy for most users, but flexible for users who have specific needs.

[2] https://pypi.python.org/pypi/PasteScript/
[3] http://www.cheetahtemplate.org/
[4] http://jinja.pocoo.org/
[5] http://docs.djangoproject.com
[6] https://pypi.python.org/pypi/collective.generic.webbuilder/
[7] http://docs.zope.org/zope.component/zcml.html

Cookiecutter

cookiecutter [8] provides command line tools to generate files using templates.

Here are some differences with *diecutter*:

cookiecutter	diecutter
Provides command line interface.	Provides WSGI application and REST API. Use things like `curl` if you like a CLI.
You need it installed on your machine.	You just need a web client.
Easily develop templates on your machine.	To develop templates, you need to install and run a local *diecutter*.
Reads templates from Github.	Reads templates from Github, anywhere in the repository.
One template per repository. Repository is dedicated to templates	Unlimited number of templates per repository. Templates are just files and directories in a filesystem.
Supports Bitbucket too.	No support for Bitbucket at the moment. Help is welcome ;)
A "cookiecutter" is a set of templates. Renders everything.	Renders directories or any part of it (sub-directories, single files).
In templates, variables are like `{{ cookiecutter.YOUR_VARIABLE }}` so templates are tied to *cookiecutter*.	No prefix for variables. Works with templates not made for *diecutter*.
Python API.	Python API lives in piecutter [1].
Shell is the user interface.	HTML forms provide a nice, human-readable and highly customizable user interface.
Only Jinja2. No plans for other template engines.	Supports several template engines. Could support even non-Python engines.

Voodoo

See https://pypi.python.org/pypi/Voodoo

And...

See also projects listed as cookiecutter alternatives [9].

Puppet, Chef and other provisioners

Provisioning tools, such as Puppet, Chef or Salt (just to name a few), use templates to generate configuration. They also do many other things.

Diecutter only deals with templates and file generation.

Templates and file generation aren't only a matter of provisioning. There are many situations where you need templates, and where provisioners would be overkill.

Diecutter also tries to make it easier to migrate from one provisioner to another. Provisioners are often tied to one template engine, one layout and one deployment process. Those items differ from one provisioner to another. With diecutter, which supports several template engines, you could easily use the same templates whatever the provisioner or deployment procedure.

[8] https://pypi.python.org/pypi/cookiecutter
[9] http://cookiecutter.readthedocs.io/en/latest/readme.html#similar-projects

References

4.4.3 License

4.4.4 Authors and contributors

- Benoît Bryon <benoit@marmelune.net>
- Rémy Hubscher <remy.hubscher@novapost.fr>
- Korantin Auguste <contact@palkeo.com>

4.4.5 Changelog

This document describes changes between each past release. For information about future releases, check milestones [1] and Vision.

0.8 (unreleased)

- Nothing changed yet.

0.7.1 (2014-07-10)

Bugfix release.

- Bug #112 - Fixed Sphinx configuration, fixed builds on readthedocs.org.

[1] https://github.com/diecutter/diecutter/issues/milestones

0.7 (2014-07-10)

Added support for multiple template engines, reviewed documentation.

- Feature #77 - One *diecutter* server can support multiple template engines:
  - The client can ask *diecutter* to use a specific engine with `engine` in query string.
  - The list of supported engines in displayed in API index.
  - Supported engines and their implementation can be configured server-side.
- Feature #104 - Get in touch via IRC: #diecutter on freenode.
- Bug #103 - README is rendered as HTML on PyPI (was plain reStructuredText).
- Refactoring #107 - Functional tests moved to Sphinx documentation.
- Refactoring #108 - *diecutter* no longer requires *webtest* and *mock*. They are only required to run tests or building documentation. Removed `diecutter.tests` from distributed package.
- Refactoring #110 - From demo, removed templates that are now maintained as third-party projects: Django project, Python project and Sphinx documentation.
- Features #81 and #87 - Reviewed documentation about alternatives and related projects: added *Voodoo* and *cookiecutter* to alternatives ; highlighted *piecutter* as the core Python API provider.

0.6 (2014-04-13)

Moved some parts of diecutter to external projects: core in piecutter, demo in template index and standalone template repositories.

- Moved generic bits of diecutter's core to third-party project 'piecutter'.
- Bug #100 - Files in tar.gz archives have a valid modification time (was epoch).
- Feature #97 - Refactored diecutter's demo index, so that it references templates from external repositories.
- Render archive filename using filename_engine.
- Added example configuration for cookiecutter and github.
- Improved contributor guide.
- Using tox to run tests. Development environment no longer uses zc.buildout.

0.5 (2013-07-19)

Proof of concept implementation of remote template loader and Django template engine.

- Features #27 and #28 - Experimental support of remote templates, where templates are hosted in Github public repositories. The github service is published at http://diecutter.io/github. It accepts URLs like http://diecutter.io/github/<owner>/<project>/<revision>/<path/to/template/resource>
- Features #57 and #29 - Public online SAAS at http://diecutter.io
- Feature #66 - Introduced support of Django template engine. You can choose the (unique) template engine with `diecutter.template_engine` configuration directive. Default value is `piecutter.engines.jinja:Jinja2Engine`.
- Bug #60 - PyPI renders README as HTML (was plain text).
- Bug #65 - Contributor guide mentions dependency to virtualenv (was missing).

- Refactoring #68 - Code follows strict PEP8. Using flake8 in tests.

0.4 (2013-07-17)

New feature (tar.gz archives) and marketing (talks).

- Feature #4 - Added support of "accept" header for POST requests on directories: accepted types are ZIP (`application/zip`) and TAR.GZ (`application/gzip`).
- Feature #53 - GZIP is now the default archive format when rendering directories. Use "diecutter.default_archive_type = application/zip" in configuration file if you need ZIP format as a default.
- Refactoring #55 - Dropped support of Python 2.6. Tests are run against Python 2.7 only.
- Refactoring #20 - Render functions return generator ; moved response composition (file/archive) into views via writers.
- Feature #46 - Added content of talks in documentation: AFPY event and EuroPython 2013.
- Feature #58 - Highlighted roadmap and vision in README.

See also milestone 0.4 on bugtracker [2].

0.3 (2013-04-16)

New features, documentation, bugfixes.

- Bug #44 - Accepted arrays in URL-encoded POST.
- Bug #40 - Setup CORS to allow AJAX requests on diecutter's API.
- Refactoring #37 - Used Jinja's environment.
- Bug #34 - Frozen buildout configuration file for development environment.
- Features #31 and #43 - Published diecutter's demo online. Online API URL changed.
- Feature #24 - Added Sphinx documentation template in diecutter's demo.
- Feature #23 - Added diecutter's Sphinx documentation.
- Feature #10 - Added dynamic tree template.

See also milestone 0.3 on bugtracker [3].

0.2 (2013-02-22)

Maintenance release, implementation refactoring, tests.

- Refactoring #22 - Added tests.
- Bug #17 - Sort directories alphabetically.
- Bug #13 - Fixed "diecutter.readonly" which was always `True`.

See also milestone 0.2 on bugtracker [4].

[2] https://github.com/diecutter/diecutter/issues?milestone=7&state=closed
[3] https://github.com/diecutter/diecutter/issues?milestone=6&state=closed
[4] https://github.com/diecutter/diecutter/issues?milestone=2&state=closed

0.1 (2013-01-29)

Initial release.

- Bug #11 - On POST requests, handle empty content-type as "application/x-www-form-urlencoded".
- Feature #8 - Support INI files as input for POST requests.
- Feature #3 - Use a configuration file outside diecutter's code.
- Feature #2 - If "readonly" option is `True`, forbid PUT requests.
- Feature #1 - Pass a "diecutter" context variable to templates, containing data such as "diecutter.api_url", "diecutter.version" and "diecutter.now".
- Feature - Diecutter service renders directories as ZIP archives.
- Feature - Diecutter service renders files.

See also milestone 0.1 on bugtracker [5].

Notes & references

4.5 Contributing to diecutter

This document provides guidelines for people who want to contribute to the project.

4.5.1 Resources

- Code repository: https://github.com/diecutter/diecutter
- Bugtracker: https://github.com/diecutter/diecutter/issues
- Continuous integration: https://travis-ci.org/diecutter/diecutter

4.5.2 Create tickets

Please use the bugtracker [1] **before** starting some work:

- check if the bug or feature request has already been filed. It may have been answered too!
- else create a new ticket.
- if you plan to contribute, tell us, so that we are given an opportunity to give feedback as soon as possible.
- Then, in your commit messages, reference the ticket with some `refs #TICKET-ID` syntax.

4.5.3 Use topic branches

- Work in branches.
- Prefix your branch with the ticket ID corresponding to the issue. As an example, if you are working on ticket #23 which is about contribute documentation, name your branch like `23-contribute-doc`.
- If you work in a development branch and want to refresh it with changes from master, please rebase [2] or merge-

[5] https://github.com/diecutter/diecutter/issues?milestone=1&state=closed
[1] https://github.com/diecutter/diecutter/issues
[2] http://git-scm.com/book/en/Git-Branching-Rebasing

based rebase [3], i.e. do not merge master.

4.5.4 Fork, clone

Clone *diecutter* repository (adapt to use your own fork):

```
git clone git@github.com:diecutter/diecutter.git
cd diecutter/
```

4.5.5 Setup a development environment

System requirements:

- Python [4] version 2.7 (in a virtualenv [5] if you like).
- make and wget to use the provided *Makefile*.

Execute:

```
git clone git@github.com:diecutter/diecutter.git
cd diecutter/
```

4.5.6 Usual actions

The *Makefile* is the reference card for usual actions in development environment:

- Install development toolkit with pip [6]: `make develop`.
- Run tests with tox [7]: `make test`.
- Build documentation: `make documentation`. It builds Sphinx [8] documentation in *var/docs/html/index.html*.
- Release *diecutter* project with zest.releaser [9]: `make release`.
- Cleanup local repository: `make clean`, `make distclean` and `make maintainer-clean`.

Notes & references

4.6 Demo

Let's try diecutter!

[3] http://tech.novapost.fr/psycho-rebasing-en.html
[4] http://python.org
[5] http://virtualenv.org
[6] https://pypi.python.org/pypi/pip/
[7] https://pypi.python.org/pypi/tox/
[8] https://pypi.python.org/pypi/Sphinx/
[9] https://pypi.python.org/pypi/zest.releaser/

4.6.1 Online SAAS

There is an online server hosting *diecutter*:

- index page: http://diecutter.io/
- demo templates API: http://diecutter.io/api/
- API using Github loader: http://diecutter.io/github/

In most cases, http://diecutter.io/ and Client howto reference should be enough for you to discover *diecutter*.

4.6.2 In sourcecode

The `demo/` directory in diecutter's sourcecode [1] contains templates in `templates/` folder. They are basically used for documentation (doctests).

Feel free to use it as a sandbox.

4.6.3 Local demo server

Here are instructions to run *diecutter*'s demo on your machine.

System requirements:

- Python [2] version 2.7, available as `python` command.

  Note: You may use Virtualenv [3] to make sure the active `python` is the right one.

- make and wget to use the provided `Makefile`.

Execute:

```
git clone git@github.com:diecutter/diecutter.git
cd diecutter/
make develop
make serve
```

The last command runs diecutter service on localhost, port 8106. Check it at http://localhost:8106/

Tip: If you cannot execute the `Makefile`, read it and adapt the few commands it contains to your needs.

4.6.4 Examples

Examples using the demo's templates are explained in Client howto.

[1] https://github.com/novagile/diecutter/
[2] http://python.org
[3] http://virtualenv.org

References

4.7 Presentations

Here are some presentations about diecutter:

- Lightning talk, at an Afpy event, Paris, march 2013 (source)
- Poster, at EuroPython, Florence, july 2013 (source)
- Lightning talk, at Write The Docs, Budapest, march 2014 (source)

4.8 FAQ

Here are some frequently asked questions...

4.8.1 Why "diecutter" name?

A "die cutter" is a machine that cuts materials to produce shapes, from molds.

Here, the diecutter web service produces files from data and templates.

4.8.2 Does it replaces my web framework (Django) templates?

diecutter is not meant to replace web framework's template engines! Diecutter is meant to replace file generators for configuration or scaffolding use cases.

Even if processes are really close, environment and needs aren't. As an example, diecutter service won't be as efficient as web frameworks internal templating systems.

So, even if, in theory, you could use diecutter as a template engine to render pages of a website, it is discouraged.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search