Optimus Documentation

Release 1.0.1

David THENON

Jun 07, 2018

Contents

Optimus is a static site builder, it produces HTML from templates (through Jinja2) with assets management (through webassets) and i18n support for translations (through Babel).

Read the Optimus documentation on https://optimus.readthedocs.org/

CHAPTER 1

Links

- Read the documentation on Read the docs;
- Download its PyPi package;
- Clone it on its Github repository;

CHAPTER 2

User's Guide

2.1 Install

You will have to install pip and virtualenv on your system. You should first install pip package then it will be easier to install virtualenv with it, like this :

```
sudo pip install virtualenv
```

It is recommended to install it in a virtualenv environment like this :

```
virtualenv --no-site-packages my_optimus_projects
cd my_optimus_projects
source bin/activate
pip install Optimus
```

This way you can work safely on your projects within this environment without any change to your system.

Also you can install it directly on your system without virtualenv, just use pip :

```
sudo pip install Optimus
```

2.1.1 Operating system

Optimus has been developed for Linux systems, it works also on Windows and MacOSX but you should have some tasks that will differs from the present documentation.

2.1.2 Asset filters

Asset filters are used to process assets, mostly to compress them.

Default install comes without any compressor requirement. It is up to you to choose, install and use assets compressors in your asset bundles.

See webassets filters documentation for more details.

2.1.3 Webserver for development

You can install cherrypy, a simple Web server, to see your builded pages :

```
pip install CherryPy
```

Read *Web server* to see how to use it.

2.1.4 Enable i18n support

Then you will have to enable it by adding the Jinja2 i18n extension in your settings :

```
JINJA_EXTENSIONS = (
    ...
    'jinja2.ext.i18n',
    ...
)
```

This is only for a new project manually created, `i18n` project template already installs this for you.

2.2 Basics

Optimus is usable with a command line tool to build pages, create new projects or enter in a watch mode that automatically rebuilds pages when their templates has been changed.

It works a little bit like Django as you create a project with a **settings** file containing all useful global settings to build your pages and manage your assets.

2.2.1 Settings

This is where your environment configuration resides, generally the `settings.py` is the default settings used in development, and the `prod_settings.py` file is used for a production environment that it inherits from the default settings and only sets a `DEBUG = False` to avoid the debug mode and minify the assets.

Optimus command line actions always accept a `settings` option to specify a settings file, by default this is the `settings.py` that is used but if you want to use another settings file like `prod_settings.py` you have to specify it in command line like a Python path :

```
optimus-cli [ACTION] --settings-name=prod_settings
```

If you just want to use the default settings, you don't need to specify it with `settings-name` option.

Below is a list of all available settings, but not all are created in the settings file when you create a new project with Optimus, only the useful ones. Optionnal settings that are undefined will be set with a default value. When the default value is not defined in the list, you can assume than they are empty.

DEBUG When setted to `True`, webassets won't try to pack and compress any bundles. This is the preferred method when developping your templates and CSS and this is why it is the default behavior in the default settings file. You should set it to `False` for production settings. This variable is available in templates context.

PROJECT_DIR Absolute path to the project directory. The settings files provided in project templates already fills them automatically, you should not need to edit it.

SITE_NAME The project name to use in your templates.

SITE_DOMAIN The project hostname (without http protocol prefixe) to use in your templates.

SOURCES_DIR Absolute path to the project sources (templates, assets, etc..) directory.

TEMPLATES_DIR Absolute path to the project templates directory.

PUBLISH_DIR Absolute path to the directory where to publish pages and assets. Don't use the same path for different settings file.

STATIC_DIR Absolute path where will be moved all the static files (from the sources), usually this is a directory in the `PUBLISH_DIR`

LOCALES_DIR Absolute path to the i18n translation catalogs directories.

WEBASSETS_CACHE The directory where webassets will store his cache. You can set this to `False` to not use the cache, or set it to True to use the default directory from webassets.

LANGUAGE_CODE Language locale name to use as the default for Pages that don't define it, see http://www.i18nguy.com/unicode/language-identifiers.html

LANGUAGES A list of locale name for all available languages to manage with PO files. Remember to add it the locale name for the default language from `LANGUAGE_CODE`.

Sample :

```
LANGUAGES = (LANGUAGE_CODE, 'fr_FR')
```

This will add the default language and French to the known languages to manage.

Sometime it is also needed to have a label for these languages or some other associated parameters, so your languages entries can be tuples but their first item **must** be the locale name. Here is a sample :

```
LANGUAGES = (
    (LANGUAGE_CODE, "International"),
    ('fr_FR', "France"),
)
```

Note that Optimus didn't care about other items in tuples of languages entries, you can add everything you want. But take care that Optimus will allways assume the first item is the locale name it needs.

STATIC_URL The static url to use in templates and with webassets. This can be a full URL like `http://`, a relative path or an absolute path.

RST_PARSER_SETTINGS ReSTructuredText parser settings to use when building a RST document. This is only useful if you use RST documents in your pages.

Default value is :

```
RST_PARSER_SETTINGS = {
    'initial_header_level': 3,
    'file_insertion_enabled': True,
    'raw_enabled': False,
    'footnote_references': 'superscript',
    'doctitle_xform': False,
}
```

BUNDLES Custom bundles to use for managing assets.

Sample :

```
BUNDLES = {
    'my_css_bundle': Bundle(
        'css/app.css',
        filters=None,
        output='css/app.min.css'
    ),
    'my_js_bundle': Bundle(
        'js/app.js',
        filters=None,
        output='js/app.min.js'
    ),
}
```

See webassets bundle documentation for more details.

ENABLED_BUNDLES Key names of enabled bundles to use, by default all knowed bundles (from setting `BUNDLES`) are enabled. If you don't want to enable them all, just define it with a list of bundle names to enable.

FILES_TO_SYNC Sources files or directories to synchronize within the published static directory. This is usually used to put on some assets in the static directory like images that don't need to be compressed with assets bundles.

Note that you should be carefull to not conflict with files targeted by webassets bundles.

JINJA_EXTENSIONS Comment, uncomment or add new extension path to use with Jinja here.

Default value is :

```
JINJA_EXTENSIONS = (
    'jinja2.ext.i18n',
)
```

Note that you don't need to manually define the webassets extension if you use it, it is automatically appended within the build process if it detects bundles.

PAGES_MAP Python path to the file that contains pages map, this is relative to your project, default value is `pages`, meaning this will search for `pages.py` file in your project directory.

I18N_EXTRACT_MAP Map for translation rules extraction with Babel.

Default value is :

```
I18N_EXTRACT_MAP = (
    ('pages.py', 'python'),
    ('*settings.py', 'python'),
    ('**/templates/**.html', 'jinja2'),
)
```

So the default behavior is only to search for translations in template sources, `pages.py` and all common settings files.

I18N_EXTRACT_SOURCES List of path to search for translation to extract. In these paths, a scan will be done using the rules from `I18N_EXTRACT_MAP`.

Default value is :

```
I18N_EXTRACT_SOURCES = (
    PROJECT_DIR,
)
```

So it will search recursively in the project directory.

I18N_EXTRACT_OPTIONS Options for translation rules extraction with Babel.

Default value is :

```
I18N_EXTRACT_OPTIONS = {
    '**/templates/**.html': {
        'extensions': 'webassets.ext.jinja2.AssetsExtension',
        'encoding': 'utf-8'
    }
}
```

2.2.2 Templates

The templates are rendered to pages using template engine Jinja2.

For each template the default context variables are :

- `debug` : A boolean, his value comes from `settings.DEBUG`;
- `SITE` : A dict containing some variables from the settings;
  - `name` : the value from `settings.SITE_NAME`;
  - `domain` : the value from `settings.SITE_DOMAIN`;
  - `web_url` : the value from `settings.SITE_DOMAIN` prefixed by *http://*;
- `STATIC_URL` : A string, containing the value from `settings.STATIC_URL`;

Read the Jinja2 documentation for more details on the available template markups.

2.2.3 Assets

You can simply put your assets where you want in the `sources` directory and add your assets directories in `settings.FILES_TO_SYNC`, they will be copied to your build directory.

But with Optimus this is only required for *real* static assets like images. For CSS and Javascript you should manage them with webassets that is already installed with Optimus.

With webassets you manage your assets as packages named `Bundle`, like a bundle for your main CSS, another for your IE CSS hacks/patchs and another for your Javascripts files. You will have to register your custom bundles in `settings.BUNDLES` and enable them in `settings.ENABLED_BUNDLES`.

The benefit of webassets is that it can pre and post process all your assets. This is usually used to *minify* and pack multiple files in one final file. Read the webassets documentation for more details how to use this and to manage bundle assets in your templates.

2.2.4 Pages

The pages to build are registred in a `pages.py` file in your project, it must contains a `PAGES` variable that is a list containing `optimus.builder.pages.PageViewBase` instances.

A default project created from the `init` (*Create a project*) command is already shipped with a `pages.py` containing some samples pages, you can change them, inherit them or add another to build various pages.

Page context

Default `PageViewBase` instance adds some variables to its template context (*Templates*) :

- **page_title** that contains the value of `PageViewBase.title` attribute;
- **page_destination** that contains the value of `PageViewBase.destination` attribute;
- **page_relative_position** that contains the relative path position from the destination file to the root of the publish directory;
- **page_lang** that contains the value of `PageViewBase.page_lang` attribute;
- **page_template_name** that contains the value of `PageViewBase.template_name` attribute;

See `optimus.builder.pages` to see more detail on how it works.

Defining your pages

There are three required arguments for a `PageViewBase` object :

title The title of your page, can be anything you want, it's just a context variable that you can use in your templates.

destination Destination file path where the page will be builded, the path is relative to the setting `PUBLISH_DIR`. You can use multiple subdirectory levels if needed, the builder will create them if it does not allready exists.

template_name File path for the template to use, the path is relative to the setting `TEMPLATES_DIR`.

The short way is like so :

```
from optimus.builder.pages import PageViewBase
# Enabled pages to build
PAGES = [
    PageViewBase(title="My page", template_name="mypage.html", destination="mypage.
↪html"),
]
```

But it is more likely you need to build more than one pages and generally you want to share some attributes like templates or title. So instead of directly using `PageViewBase`, you should make your own page object like this :

```
from optimus.builder.pages import PageViewBase

class MyBasePage(PageViewBase):
    title = "My base page"
    template_name = "mypage.html"

# Enabled pages to build
PAGES = [
    MyBasePage(title="My index", destination="index.html"),
    MyBasePage(title="My Foo page", destination="foo.html"),
    MyBasePage(title="My Bar page", destination="bar.html"),
]
```

Extending PageViewBase

You can override some methods to add logic or change some behaviors in your `PageViewBase` object.

PageViewBase.get_title Set the `page_title` context variable.

PageViewBase.get_destination Set the `page_destination` context variable.

PageViewBase.get_relative_position Set the `page_relative_position` context variable.

PageViewBase.get_lang Set the `page_lang` context variable.

PageViewBase.get_template_name Set the `page_template_name` context variable.

PageViewBase.get_context Set the context page to add variables to expose in the templates. The method does not attempt any argument and return the context.

To add a new variable `foo` in your context you may do it like this :

```
class MyPage(PageViewBase):
    title = "My page"
    template_name = "mypage.html"
    destination = "mypage.html"

    def get_context(self):
        # This line set the default context from PageViewBase
        super(MyPage, self).get_context()
        # Add your new variables here
        self.context.update({
            'foo': 'bar',
        })
        return self.context
```

2.2.5 Translations

Marked strings with the `{% trans %}` template tag in your templates (see Jinja2 template documentation) will be translated from the page locale name and its associated translation catalog. They will be extracted and stored in catalog files where you will have to fill the translations. Then compile your catalog files and then, the page building will replace strings with the translation accordingly to the page language.

The recommended way is to use the Optimus command `po` see this in *Managing translations*.

Pages language

By default, Pages use a default locale language that is *en_US*, for each language you will need to make a page view with the wanted language. You can specify it in the **lang** page attribute, or in a `lang` argument when you instanciate your `PageViewBase`.

Managing translation catalog with the raw way

The *raw* way is to directly use Babel command line tool, you will have many more option to manage your catalogs but you will have to use many different commands and paths.

Before building your internationalized Pages, you will have to create a messages catalog for each needed language. Put all your `{% trans %}` tags in your templates, then make a catalog from the extracted string.

To correctly extract all your strings to translate, Babel will need some rules to know what and where it should search. This is done in a Babel mapping file, generally as a `babel.cfg` in the root directory of your project.

At least, you will need the Jinja2 integration rule :

```
[jinja2: sources/templates/**.html]
encoding = utf-8
extensions = webassets.ext.jinja2.AssetsExtension
```

The last line is needed if you use webassets tags `{% assets %}...{% endassets %}` in your templates, otherwise the extraction will fail. See the Jinja2 integration documentation for more details.

Extracting first the reference POT file :

```
pybabel extract -F babel.cfg -o locale/messages.pot .
```

Initialize the language files (repeat this for each needed language with his correct locale key) :

```
pybabel init -l en_US -d locale -i locale/messages.pot
```

Compile all your language files :

```
pybabel compile -f -d locale
```

Update them when you make changes in your template strings (after this, you'll need to re-compile them) :

```
pybabel update -l en_US -d locale -i locale/messages.pot
```

2.3 Usage

You can use Optimus from the command line tool `optimus-cli`. A global help is available with :

```
optimus-cli -h
```

Or specific command action help with :

```
optimus-cli COMMAND -h
```

There is also a common command argument `--settings-name` that is useful to define the settings files to use. It appends a Python path to the settings file. For common usage you just have to give the filename without the `.py` extension, otherwise you will get an error message.

2.3.1 Create a project

At least you will give a name for the new project. Be aware that it must a valid Python module name, so only with alphanumeric characters and _. No spaces, no dots, etc.. :

```
optimus-cli init my_project
```

It will create project directory and fill it with basic content. There is some sample project templates:

- `basic` : Default template, you don't have to specify anything to use it;
- `i18n` : An i18n sample template. All needed stuff to enable i18n support are installed;

All project template includes a `Makefile` to ease install and usage, CherryPy is installed by default and Assets settings use `rcssmin` and `jsmin`.

To create a new project with the I18n sample, you will have to do something like :

```
optimus-cli init my_project -t i18n
```

2.3.2 Building

Configure your settings if needed, then your Pages to build and finally launch Optimus to build them :

```
optimus-cli build
```

2.3.3 Managing translations

Optimus can manage your translations for the known languages of your project. This is done in the setting `LANGUAGES` where you define a list of locale names, each of which will have a translation catalogs after you initialize them. By default, this settings is only filled with the default locale defined in the settings `LANGUAGE_CODE`. This is your responsability to fill the setting `LANGUAGES` with valid locale names.

Assuming you want to add French translations, you will have to add this setting :

```
# A list of locale name for all available languages to manage with PO files
LANGUAGES = (LANGUAGE_CODE, 'fr_FR')
```

Note the first item that also adds the locale name from the default language from the setting `LANGUAGE_CODE`.

Then you will need to flag the strings to translate in your templates with the `{% trans %}` template tag from Jinja2 (see Jinja2 template documentation for more details) like this :

```
<html>
<body>
    <h1>{% trans %}Hello world{% endtrans %}</h1>
</body>
</html>
```

And finally manage your translation catalogs, see below.

Initialize

On a new project you have to initialize the catalog template (the source used to create or update translation catalogs, represented by a `*.POT` file in your locales directory) :

```
optimus-cli po --init
```

This will extract translation strings from your templates (and other files in your sources directory if needed) and put them in catalog templates, then after translation catalogs will be created from the template for each knowed languages.

This command is safe for existing translations, if a translation catalogs allready exists, it will not be overwrited. Only non existing translation catalogs will be created.

Now open your catalog files (`*.PO`) edit them to fill the translations for your languages, then compile them (see *Compilation*).

Update

If you do some changes on translations in your templates, like add new translation strings, modify or remove some, you have to update your catalogs to adapt to this changes :

```
optimus-cli po --update
```

This will extract again your translation strings, update the catalog template then update your translation catalogs. After that you will have to re-compile them (see *Compilation*).

Compilation

Catalog files (`*.PO`) are not usable for page building, you will have to compile them first, this is done with the command line :

```
optimus-cli po --compile
```

It will compile the catalog file to `*.MO` files, this way Optimus can use your translations. Remember that when you do updates on catalog files you will have to re-compile them each time, this is not automatic.

Note that also when you edit your translation catalogs to change some translations, you will have to re-compile them.

2.3.4 Watch mode

Use the `watch` command action to automatically rebuild files at each change in your sources :

```
optimus-cli watch
```

This will launch a process that will watch for changes and rebuild pages if needed. For changes on templates, the watch mode will only rebuild pages that uses the changed templates. Also if it detects that the publish directory (from the setting `PUBLISH_DIR`) does not exists, it will automatically performs a first build.

To stop the watcher process, just use the common keyboard combo `CTRL+C`.

This is useful in development, but note that the watcher is limited to watch only for templates and assets changes.

Watch mode will not detect if :

- You change some things in your Page views, your settings or your RST files;
- You add new static files;
- You make some changes in your translation files (`*.pot` and `*.po`);

For theses cases you will have to stop the watcher, manually rebuild with `build` command or Babel tool (for translations only) then relaunch the watcher.

2.3.5 Web server

You can launch a simple web server to publish your builded content, **it's not intended to be used in production**, only for debugging your work. This command action is only available if you already have installed **cherrypy**, see the *Install* document about this.

The hostname argument is required and it should at least contain the port (like '80'), the default address will be "127.0.0.1" if you don't give it.

To launch the webserver binded on your local IP on port 8001 to publish your project from the default settings, do this :

```
optimus-cli runserver 0.0.0.0:8001
```

Also you can bind it on localhost on port 8080 with the production settings :

```
optimus-cli runserver localhost:8080 --settings-name=prod_settings
```

The settings are used to know the publish directory to expose.

Note that the server does not build anything, it only expose the publish directory to publish the builded page and static files it contains. You should launch the *Watch mode* in parallel.

CHAPTER 3

Developer's Guide

3.1 Development

3.1.1 Development requirement

Optimus is developed with:

- *Test Development Driven* (TDD) using Pytest;
- Respecting flake and pip8 rules using Flake8;
- Sphinx for documentation with enabled Napoleon extension (using only the *Google style*);

Every requirement is available in file `requirements/dev.txt`.

3.1.2 Install for development

First ensure you have pip and `python-venv` package installed then type:

```
git clone https://github.com/sveetch/Optimus.git
cd optimus
make install-dev
```

Optimus will be installed in editable mode from the last commit on master branch.

When it's done, you will be able to check for optimus version, just type:

```
venv/bin/optimus version
```

Unittests

Unittests are made to works on Pytest, a shortcut in Makefile is available to start them on your current development install:

```
make tests
```

Tox

To ease development against multiple Python versions a tox configuration has been added. You are strongly encouraged to use it to test your pull requests.

Before using it you will need to install tox, it is recommended to install it at your system level (tox dependancy is not in tests requirements file):

```
sudo pip install tox
```

Then go in the `optimus` module directory, where `the setup.py` and `tox.ini` live and execute tox:

```
tox
```

Documentation

sphinx-autobuild is installed for a watcher which automatically rebuild HTML documentation when you change sources.

When environnement is activated, you can use following command from `docs/` directory:

```
make livehtml
```

And go on `http://127.0.0.1:8002/`.

3.2 Changelog

3.2.1 Version 1.0.1 - 2018/06/07

- Fixed documentation;
- Fixed project templates Makefile;

3.2.2 Version 1.0.0 - 2018/06/07

Rewriting everything to be Python >=2.7 and Python3 compatible with unittests coverage using pytest and tox.

- Drop 'argh' in favor of 'click' for commandline scripts, this involve commandline has a minor changes on command options usage, close #23;
- Big cleaning for sanity and update for Python3 support, close #22;
- Support of rcssmin filter for assets;
- ReStructuredText view has been dropped;
- Your old projects should still be compatible minus some specific settings details;

3.2.3 Version 0.8.2 - 2017/01/15

- Relaxed `webassets` version requirement since the last one (0.12.1) has been validated;
- Removed `yuicompressor` requirement. ClosureJS is recommended for Javascript compression since YUI is not maintained anymore. But finally Optimus do not require anymore any compressor library. It's up to the user choice;
- Removed `EXTRA_BUNDLES` occurrences since it was deprecated long time ago;
- Updated documentation;

3.2.4 Version 0.8.1 - 2017/01/01

- Validated working with `CherryPy==8.7.0`, so remove every occurences about 3.x.x version;
- Better README/Doc index/Package short description;

3.2.5 Version 0.8.0 - 2016/12/31

- Include `html5writer.py` taken from `rstview` and so remove dependency to `rstview`, close #19;
- Move changelog to its own file, updated documentation Makefile, added dev requirements;
- Use `sphinx_rtd_theme` in documentation if available;
- Improved watcher logging output a little bit so it reveals changed file when detected without to use the debug level;
- Do not enable anymore `runserver` command to installed CherryPy, instead raise a better error message explanation;

3.2.6 Version 0.7.2 - 2016/05/05

Minor update that modify 'settings' and 'pages' modules import so exception is raised to ease debugging.

3.2.7 Version 0.7.1 - 2015/06/14

Dummy release just to update documentation about forgotten changelog.

3.2.8 Version 0.7.0 - 2015/06/14

- Upgraded dependancy to watchdog==0.8.3 to try to fix a problem with watch mode on OSX;
- Fixed doc;
- Changed module imports to have distinct error name for page and settings import errors;
- Changed message error for module loading to be more helpful;

3.2.9 Version 0.6.9

- Fix a bug with bad signature for `po` command;
- Moving script name from **optimus** to **optimus-cli** because this was causing issues with `setup.entry_points` usage and buildout;

3.2.10 Version 0.6.8.1

Update Argh dependancy to `>= 0.24.1`.

3.2.11 Version 0.6.8

Re-use a fixed version for **argh** because the 0.24 version has incompatible backward issues.

3.2.12 Version 0.6.7.1

Fix dependancies syntax in setup.py that was causing issues during installation.

3.2.13 Version 0.6.7

- Remove CherryPy dependancy from setup.py, add an install note about this;
- Update documentation;

3.2.14 Version 0.6.6

Upgrade to yuicompressor 2.4.8

3.2.15 Version 0.6.5

Updating doc, in setup.py use 'entry_points' instead of 'scripts'

3.2.16 Version 0.6.4

- Fixing update method in po command to update the POT file;
- Add I18N_EXTRACT_SOURCES setting and use it in extraction method, bumping version;
- Add new behavior for settings.LANGUAGES to permit tuples instead of simple locale name;

3.2.17 Version 0.6.1

- Setting name `EXTRA_BUNDLES` is deprecated and **will be removed in a futur release**. In project settings rename it to `BUNDLES`;
- Remove `optimus.builder.assets.COMMON_BUNDLES`, this was containing default bundles that was not really useful. If your project used them, you will have errors on page building about missing bundles, you can recover them in your `settings.BUNDLES` from :

```
COMMON_BUNDLES = {
    'css_screen_common': Bundle(
        'css/screen.css',
        filters='yui_css',
        output='css/screen.min.css'
    ),
    'css_ie_common': Bundle(
        'css/ie.css',
        filters='yui_css',
        output='css/ie.min.css'
    ),
    'js_ie_common': Bundle(
        'js/modernizr.custom.js',
        'js/respond.src.js',
        filters='yui_js',
        output='js/ie.min.js'
    ),
    'js_jquery': Bundle(
        'js/jquery/jquery-1.7.1.js',
        filters='yui_js',
        output='js/jquery.min.js'
    ),
}
```

3.2.18 Version 0.6 - 2013/12/16

- Add new command `po` to automatically manage translations files;
- Add better error messages for some command line options;
- Add a required settings list that is checked when loading settings file to avoid error on missing settings;
- Add default values to un-required settings so the settings file is more clean and short with only needed settings;
- Now Babel, cherrypy and yui-compressor are required dependancies;
- The previous commande line tool name `optimus-cli` has been chaned to a more shorter name `optimus`;
- New settings have been added to manage languages and translations with the new command `po`;
- Settings files have been simplified, making some settings optionnal to have a more clean and short settings files;
- `watch` command options : automatically perform the first build when the build directory does not exits to avoid errors with the watcher;
- `init` command options : `--name` has moved to a positionnal argument;
- Project templates : Removed requirements.txt for pip since the `setup.py` contains all needed stuff;
- Project templates : Renamed “sample” to “basic” and “sample_i18n” to “i18n”. Also add aliases for them, so you just have to use their names and not anymore their full Python paths;
- Project templates : Changing to better templates with assets, SCSS sources and Compass config;

3.3 Core API

Optimus is mainly a commandeline tool but it relies on a core API that may be used from another application.

3.3.1 Modules

Exceptions

Specific exceptions that Optimus code can raise.

exception optimus.exceptions.**DestinationExists**
Bases: *optimus.exceptions.OptimusBaseException*

Exception to be raised when a destination allready exists for a new project to create.

exception optimus.exceptions.**InvalidHostname**
Bases: *optimus.exceptions.OptimusBaseException*

Exception to be raised when a parsed hostname is invalid.

exception optimus.exceptions.**InvalidLanguageIdentifier**
Bases: *optimus.exceptions.OptimusBaseException*

Exception to be raised from `lang.LangBase` when given language code is invalid.

exception optimus.exceptions.**InvalidSettings**
Bases: *optimus.exceptions.OptimusBaseException*

Exception to be raised for invalid settings from 'conf.model'

exception optimus.exceptions.**OptimusBaseException**
Bases: `Exception`

Base for Optimus exceptions.

exception optimus.exceptions.**TemplateImportError**
Bases: *optimus.exceptions.OptimusBaseException*

Exception to be raised when a template module import fails.

exception optimus.exceptions.**TemplateSettingsInvalidError**
Bases: *optimus.exceptions.OptimusBaseException*

Exception to be raised when a template manifest have wrong settings.

exception optimus.exceptions.**ViewImproperlyConfigured**
Bases: *optimus.exceptions.OptimusBaseException*

Exception to be raised from `pages.view.base.PageViewBase` when instanciated with bad value or missing parameters.

Logging

Optimus makes a lot usage of logging during its processes to inform user about what it is doing or errors that occured.

To be more readable, its logger is configured to be colored using `colorlog` package.

optimus.logs.**init_logger**(*level*, *printout=True*)
Initialize app logger to configure its level/handler/formatter/etc..

Parameters **level** (*str*) – Level name (`debug`, `info`, etc..).

Keyword Arguments **printout** (*bool*) – If False, logs will never be outputed.

Returns Application logger.

Return type logging.Logger

Project settings

Settings model

class optimus.conf.model.**SettingsModel**
Settings model

Basically empty on init, you'll have to fill it either from kwargs or a module (or an object).

Be aware that on default behavior settings loading methods execute `apply_defaults` method that will apply default values for some settings but related settings are not updated, you need to take care of them yourself or disabled default values pratice.

_excluded_names
list – Setting names to exclude from loading methods.

_required_settings
list – Settings names required to be defined from loading methods.

_default_babel()
Set default attributes for required settings around Babel

_default_jinja()
Set default attributes for required settings around Jinja

_default_rst()
Set default attributes for required settings ReSTructuredText

_default_watchdog()
Set default attributes for required settings Watchdog

_default_webassets()
Set default attributes for required settings around Webassets

apply_defaults()
Apply default attributes for needed but not required settings.

check()
Check every required settings are defined

load_from_kwargs(*check=True*, *defaults=True*, ***kwargs*)
Set setting attribute from given named arguments.

Keyword Arguments

- **check** (*bool*) – True to perform required settings check. Default is True.
- **defaults** (*bool*) – True to set default needed settings.
- **kwargs** – Named arguments to load as settings.

Returns List of loaded setting names from given arguments.

Return type list

load_from_module(*settings_module*, *check=True*, *defaults=True*)
Set setting attribute from given module variables.

Keyword Arguments

- **settings_module** (*object*) – Object to find attributes to load as settings. Every valid attribute names will be used.
- **defaults** (*bool*) – True to set default needed settings.

- **kwargs** – Named arguments to load as settings.

Returns List of loaded setting names from given module.

Return type list

validate_name(*name*)

Filter to validate setting name

Name must be uppercase, not starting with a '_' character and not registred in exluded names.

Parameters **name** (*string*) – Setting name.

Returns True if name is valid, else False.

Return type bool

Module loader helpers

optimus.conf.loader.**import_pages_module**(*name*, *basedir=None*)

Shortcut to have specific error message when loading a page module

Parameters **name** (*str*) – Module name to retrieve from `basedir`.

Keyword Arguments **basedir** (*str*) – Base directory from where to find module name. If no base directory is given `os.getcwd()` is used. Default is `None`.

Returns: Finded and loaded module.

optimus.conf.loader.**import_project_module**(*name*, *basedir=None*, *finding_module_err='Unable to find module: {0}'*, *import_module_err='Unable to load module: {0}'*)

Load given module name.

Parameters **name** (*str*) – Module name to retrieve from `basedir`.

Keyword Arguments

- **basedir** (*str*) – Base directory from where to find module name. If no base directory is given `os.getcwd()` is used. Default is `None`.
- **finding_module_err** (*str*) – Message to output when the given module name is not reachable from `basedir`.
- **import_module_err** (*str*) – Message to output when the given module name raise exception when loaded.

Returns Finded and loaded module.

Return type object

optimus.conf.loader.**import_settings**(*name*, *basedir*)

Load settings module.

Validate required settings are set, then fill some missing settings to a default value.

Parameters

- **name** (*str*) – Settings module name to retrieve from `basedir`.
- **basedir** (*str*) – Base directory from where to find settings module name.

Returns Settings module.

Return type object

optimus.conf.loader.**import_settings_module**(*name*, *basedir=None*)
Shortcut to have specific error message when loading settings module

Parameters **name** (*str*) – Module name to retrieve from basedir.

Keyword Arguments **basedir** (*str*) – Base directory from where to find module name. If no base directory is given os.getcwd() is used. Default is None.

Returns Finded and loaded module.

Return type object

Assets registry

class optimus.assets.registry.**AssetRegistry**
Index all knowed files from registered bundles

map_dest_to_bundle
dict – Registry of asset paths associated to their asset bundle keyname.

logger
logging.Logger – Optimus logger.

add_bundle(*name*, *bundle*)
Add a bundle to the registry

Parameters

- **name** (*string*) – Bundle name as defined in the assets map.
- **bundle** (*webassets.Bundle*) – Bundle to associate to given name.

optimus.assets.registry.**register_assets**(*settings*)
Initialize webassets environment and its bundles.

Parameters **settings** (conf.model.SettingsModel) – Settings registry instance.

Returns New configured Webasset environment.

Return type webassets.Environment

class optimus.assets.rcssmin_webassets_filter.**RCSSMin**(***kwargs*)
Minifies CSS using 'rcssmin' library.

Requires the rcssmin package (https://github.com/ndparker/rcssmin).

This is a simple webassets filter that has been merged but not released yet, so we ship it until new package release since 'cssmin' is totally outdated and doesn't work with Python 3.

Watcher handlers

This component contains watchdog handlers to use to watch a project sources.

class optimus.watchers.**BaseHandler**
Base class for handlers.

Assume children inheriting this class have a settings attribute with a valid SettingsModel instance as value.

get_relative_asset_path(*path*)
Retrieve relative path from assets directory.

Parameters **path** (*str*) – Path to an asset file.

Returns Relative path either from assets directory or untouched if file does not belong to assets dir.

Return type string

get_relative_template_path(*path*)
Retrieve relative path from templates directory.

Parameters **path** (*str*) – Path to a template file.

Returns Relative path either from templates directory or untouched if file does not belong to template dir.

Return type string

class optimus.watchers.templates.**TemplatesWatchEventHandler**(*settings*, *pages_builder*, **args*, ***kwargs*)
Template events handler.

Parameters

- **settings** (optimus.conf.model.SettingsModel) – Project settings.
- **pages_builder** (optimus.pages.builder.PageBuilder) – Page builder object that is triggered to perform page building.
- **args** – Additional arguments to be passed to handler, commonly for watchdog API.

Keyword Arguments **kwargs** – Optionnal keyword arguments commonly for watchdog API.

settings
optimus.conf.model.SettingsModel – As given from arguments.

pages_builder
optimus.pages.builder.PageBuilder – As given from arguments.

logger
logging.Logger – Optimus logger.

build_for_item(*path*)
Build all pages using given template path.

If template is a snippet included in other templates they will be flagged for build too. This is recursive so a snippet in a snippet in a snippet will raises to page templates.

`path` argument is a template path

Parameters **path** (*string*) – Template path.

Returns List of builded pages.

Return type list

on_created(*event*)
Called when a new file or directory is created.

Parameters **event** – Watchdog event, either `watchdog.events.DirCreatedEvent` or `watchdog.events.FileCreatedEvent`.

on_modified(*event*)
Called when a file or directory is modified.

Parameters **event** – Watchdog event, `watchdog.events.DirModifiedEvent` or `watchdog.events.FileModifiedEvent`.

on_moved(*event*)
Called when a file or a directory is moved or renamed.

Many editors don't directly change a file, instead they make a transitional file like `*.part` then move it to the final filename.

Parameters **event** – Watchdog event, either `watchdog.events.DirMovedEvent` or `watchdog.events.FileModifiedEvent`.

class optimus.watchers.assets.**AssetsWatchEventHandler**(*settings*, *assets_env*, *pages_builder*, **args*, ***kwargs*)
Assets events handler

Since assets filename change when they are rebuiled, this handler also performs page rebuild to include the right assets urls.

Parameters

- **settings** (`optimus.conf.model.SettingsModel`) – Project settings.
- **assets_env** (*webassets.Environment*) – Webasset environment.
- **pages_builder** (`optimus.pages.builder.PageBuilder`) – Page builder object that is triggered to perform page building.
- **args** – Additional arguments to be passed to handler, commonly for watchdog API.

Keyword Arguments **kwargs** – Optionnal keyword arguments commonly for watchdog API.

settings
optimus.conf.model.SettingsModel – As given from arguments.

assets_env
webassets.Environment – Webasset As given from arguments.

pages_builder
optimus.pages.builder.PageBuilder – As given from arguments.

logger
logging.Logger – Optimus logger.

build_for_item(*path*)
Build bundle containing given asset path and all pages.

Parameters **path** (*string*) – Asset path.

Returns List of builded pages.

Return type list

on_created(*event*)
Called when a new file or directory is created.

Parameters **event** – Watchdog event, either `watchdog.events.DirCreatedEvent` or `watchdog.events.FileCreatedEvent`.

on_modified(*event*)
Called when a file or directory is modified.

Parameters **event** – Watchdog event, `watchdog.events.DirModifiedEvent` or `watchdog.events.FileModifiedEvent`.

on_moved(*event*)
Called when a file or a directory is moved or renamed.

Many editors don't directly change a file, instead they make a transitional file like `*.part` then move it to the final filename.

Parameters **event** – Watchdog event, either `watchdog.events.DirMovedEvent` or `watchdog.events.FileModifiedEvent`.

Page building and views

class optimus.pages.registry.**PageRegistry**(*elements={}*)

Page registry

Index templates and memorize page destination that use them.

Keyword Arguments **elements** (*dict*) – Initial element dictionnary. Default to an empty dict.

elements

string – Dictionnary indexed on template names which contain destinations using them.

map_dest_to_page

string – Dictionnary indexed on destinations which contain their related page view.

logger

logging.Logger – Optimus logger.

add_page(*page*, *templates*)

Add a page to registry.

Parameters

- **page** (*optimus.pages.views.PageViewBase*) – Page instance
- **templates** (*list*) – List of templates names to link to given page instance.

get_all_destinations()

Return all registered destinations

Returns List of all page destinations.

Return type list

get_all_pages()

Return all registered pages

Returns List of all page instances.

Return type list

get_pages_from_dependency(*template_name*)

Get page list depending from a template.

This method is not safe out of the context of scanned pages, because it use an internal map builded from the scan use by the add_page method. In short, it will raise a KeyError exception for every destination that is unknowned from internal map.

Parameters **template_name** (*string*) – Template name to search for.

Returns List of page instances depending from given template name.

Return type list

class optimus.pages.builder.**PageBuilder**(*settings*, *jinja_env=None*, *assets_env=None*, *dry_run=False*)

Builder class to init Jinja2 environment and build given pages.

Parameters **settings** (conf.model.SettingsModel) – Settings registry instance.

Keyword Arguments

- **jinja_env** (*jinja2.Jinja2Environment*) – Jinja2 environment. Default is `None`.
- **assets_env** (*webassets.Environment*) – Webasset environment. Default is `None`.
- **dry_run** (*boolean*) – Enable dry run mode. Default is `False`.

logger
logging.Logger – Optimus logger.

settings
conf.model.SettingsModel – Settings registry instance.

jinja_env
jinja2.Jinja2Environment – Jinja2 environment. Default is `None`.

assets_env
webassets.Environment – Webasset environment. Default is `None`.

internationalized
boolean – Indicate it internationalization is enabled. Will be automatically set to `True` if Jinja environment enable the i18n extension.

translations
dict – Dictionnary of translation catalog indexed on language identifier.

registry
optimus.pages.registry.PageRegistry – Registry of all knowed page from scanning.

dry_run
boolean – Dry run mode.

build_bulk(*page_list*)
Build all given pages.

Return all the effective builded pages

Parameters **page_list** (*list*) – List of page instances.

Returns List of destination paths from builded pages.

Return type list

build_item(*page_item*)
Build given page.

Possibly connect settings to page instance if not allready done.

Parameters **page_item** (*optimus.pages.views.PageViewBase*) – Page instance.

Returns Destination path from builded page.

Return type string

get_environnement(*assets_env=None*)
Init and configure Jinja environment.

Automatically enable some extensions and link possible asset environment.

Keyword Arguments **assets_env** (*webassets.Environment*) – Webasset environment. Default is `None`. If empty, webassets will not be available from page templates.

Returns Configured Jinja2 environment.

Return type jinja2.Jinja2Environment

get_globals()
Get global context variables from settings.

Returns Context variables.

Return type dict

get_translation_for_item(*page_item*)
Try to load the translations for the page language if any, then install it in Jinja2.

It does not reload a language translations if a previous page has allready loaded it.

Parameters **page_item** (*optimus.pages.views.PageViewBase*) – Page instance which its language identifier will be used to search for translation catalog.

Returns Translations object to give to Jinja i18n extension.

Return type babel.support.Translations

scan_bulk(*page_list*)
Scan all given pages to set their dependancies

Parameters **page_list** (*list*) – List of page instances.

Returns Every template name involved in scanned page instances.

Return type list

scan_item(*page_item*)
Scan given page to retrieve template dependancies.

Possibly connect settings to page instance if not allready done.

Parameters **page_item** (*optimus.pages.views.PageViewBase*) – Page instance.

Returns All used templates from given page.

Return type string

class optimus.pages.views.base.**PageViewBase**(***kwargs*)
Base view object for a page

You can set class attributes at the init if needed

The render method is responsible to rendering the HTML from the template and his context. Actually this is the only used method directly.

Only `lang` and `context` attributes are optional, so take care to set all the required ones because their default value is `None`. You should not use directly `PageViewBase`, inherit it in a common object with all attributes setted by default.

Template context will have the following variables :

page_title Page title

page_destination Page destination

page_lang Given langage if any

page_template_name Template name used to compile the page HTML

But you can add new variable if needed. The default context variables can not be overriden from the `context` class attribute, only from the `get_context` class method.

View need settings to be defined either as argument on instance init or later through attribute setter.

title
string – Page title.

template_name
string – Page template file path relaive to templates directoy. Used as Python template string with optional non positional argument `{{ language_code }}` available for internationalized pages.

destination
string – Page destionation path relative to build directory.

lang
string – Language identifier or an instance of `optimus.i18n.LangBase`.

context
dict – Initial page template context.

logger
logging.Logger – Optimus logger.

_used_templates
list – List of every used templates. Only filled when `introspect()` method is executed. Default to `None`.

__settings
conf.model.SettingsModel – Settings registry instance when given in kwargs. Default to `None`.

Parameters ****kwargs** – Arbitrary keyword arguments. Will be added as object attribute.

_recurse_template_search(*env*, *template_name*)
Load involved template sources from given template file path then find their template references.

Parameters

- **env** (*jinja2.Jinja2Environment*) – Jinja environment.
- **template_name** (*string*) – Template file path.

Returns List of involved templates sources files.

Return type list

get_context()
Get template context.

Returns Template context of variables.

Return type dict

get_destination()
Get page destination path.

Returns Page destination path relative to build directory.

Return type string

get_lang()
Get page language object.

Returns Language object. If `lang` page attribute is `None` it will create a language object using default language identifier from setting `LANGUAGE_CODE`.

Return type optimus.i18n.LangBase

get_relative_position()
Get relative path position from the destination file to the root.

Returns Either something like "../../" if the destination is in subdirectories or "./" if at the root. Won't never return empty string.

Return type string

get_template_name()
Get template file path.

Returns Template file path relative to templates directory.

Return type string

get_title()
Get page title.

Default behavior is to used page attribute `title`.

Returns Page title.

Return type string

introspect(*env*)
Take the Jinja2 environment as required argument to find every templates dependancies from page.

Parameters **env** (*jinja2.Jinja2Environment*) – Jinja environment.

Returns List of involved templates sources files.

Return type list

render(*env*)
Take the Jinja2 environment as required argument.

Parameters **env** (*jinja2.Jinja2Environment*) – Jinja environment.

Returns HTML builded from page template with its context.

Return type string

settings
`settings` attribute getter, check settings have been correctly defined.

Returns Settings registry instance when given in kwargs. Default to `None`.

Return type *conf.model.SettingsModel*

validate()
Validate every required attribute is set.

Returns `True` if requirements are set.

Return type boolean

Internationalization and localization

Language base object

class optimus.i18n.lang.**LangBase**(*code=None*, *label=None*)
Language base object to encapsulate the language label, code and other details.

Alternative and External code are not really used internally in optimus, there are only for some template usage.

The instance will also supply a "language_name" and "region_name" class attributes, which are the result of splitting the code on two parts. "region_name" is `None` by default, as the region name is optional in language identifier.

See http://www.i18nguy.com/unicode/language-identifiers.html for more details on language identifiers.

Usage :

```
class LangFr(LangBase):
    code = 'fr'
    label = 'France'
```

Or :

```
lang = LangBase(code="zh_CN", label="Chinese")
```

Keyword Arguments

- **code** (*string*) – Language identifier.
- **label** (*string*) – Language label like "Français" for `fr`.

label
string – Default language label if not given in kwargs.

code
string – Default language identifier if not given in kwargs.

alt_code
string – Alternative code, will be equal to "code" if not set.

external_code
string – External code for some external apps, will be equal to `alt_code` if not set.

split_code(*code*)
Split language identifier to language name and region name (if any).

Parameters **code** (*string*) – Language identifier.

Returns A pair of language name and possibly region name, if code does not contain any region it will be `None`.

Return type tuple

I18n management

I18n management support for Optimus environnment.

Only "messages.*" files for POT and PO files are managed and no other catalog type.

class optimus.i18n.manager.**I18NManager**(*settings*)
I18n manager for translation catalogs

Made to work simply within Optimus environnment, so not all of babel options are used. This way the manager can work cleanly and is more easy to use.

Parameters **settings** (`conf.model.SettingsModel`) – Settings registry instance.

catalog_name
string – Catalog filename template.

catalog_path
string – Catalog language directory template.

header_comment
string – Header comment to prepend to catalog files.

settings
conf.model.SettingsModel – Settings registry instance.

logger
logging.Logger – Optimus logger.

build_pot(*force=False*)
Extract translation strings and create Portable Object Template (POT) from enabled source directories using defined extract rules.

Note: May only work on internal '_pot' to return without touching 'self._pot'.

Keyword Arguments **force** (*boolean*) – Default behavior is to proceed only if POT file does not allready exists except if this argument is `True`.

Returns Catalog template object.

Return type babel.messages.catalog.Catalog

check_catalog_path(*locale*)
Check if a translations catalog exists

Parameters **locale** (*string*) – Language identifier.

Returns True if catalog file exists.

Return type boolean

check_locales_dir()
Check if LOCALES_DIR directory exists

Returns True if base catalog directory exists.

Return type boolean

check_template_path()
Check if the catalog template exists

Returns True if catalog template file exists.

Return type boolean

clone_pot()
Helper to clone POT catalog from writed file (not the one in memory) without to touch to `_pot` attribute.

Returns Clone catalog template object.

Return type babel.messages.catalog.Catalog

compile_catalogs(*languages=None*)
Compile PO catalogs to MO files

Note: Errors have no test coverage since `read_po()` pass them through warnings print to stdout and this is not blocking or detectable. And so the code continue to the compile part.

Keyword Arguments **languages** (*list*) – List of languages to process. Default is `None` so languages are taken from `LANGUAGES` settings.

Returns List of language identifiers for compiled catalogs.

Return type list

get_catalog_dir(*locale*)
Return the full path to a translations catalog directory

Parameters **locale** (*string*) – Language identifier.

Returns Catalog directory path.

Return type string

get_mo_filepath(*locale*)
Return the full path to a compiled translations catalog file

Parameters **locale** (*string*) – Language identifier.

Returns Compiled catalog file path.

Return type string

get_po_filepath(*locale*)
Return the full path to a translations catalog file

Parameters **locale** (*string*) – Language identifier.

Returns Catalog file path.

Return type string

get_template_path()
Return the full path to the catalog template file

Returns Catalog template file path.

Return type string

init_catalogs(*languages=None*)
Create PO catalogs from POT if they dont allready exists

Keyword Arguments **languages** (*list*) – List of languages to process. Default is `None` so languages are taken from `LANGUAGES` settings.

Returns List of language identifiers for created catalogs.

Return type list

init_locales_dir()
Create catalog base directory defined from `LOCALES_DIR` settings if it does not allready exists.

parse_languages(*languages*)
Allways return a list of locale name from languages even if items are simple string or tuples. If tuple, assume its first item is the locale name to use.

Parameters **languages** (*list*) – List of languages identifiers.

Returns Dictionnary of languages identifiers.

Return type dict

pot
Return the catalog template

Get it from memory if allready opened, if allready exists then open it, else extract it and create it.

Returns Catalog template object.

Return type babel.messages.catalog.Catalog

safe_write_po(*catalog*, *filepath*, ***kwargs*)
Safely write or overwrite a PO(T) file.

Try to write catalog to a temporary file then move it to its final destination only writing operation did not fail. This way initial file is not overwrited when operation has failed.

Original code comes from `babel.messages.frontend`.

Parameters

- **catalog** (*babel.messages.catalog.Catalog*) – Catalog object to write.
- **filepath** (*string*) – Catalog file path destination.
- ****kwargs** – Additional arbitrary keyword argumentsto pass to `write_po()` babel function.

Returns Catalog template object.

Return type babel.messages.catalog.Catalog

update_catalogs(*languages=None*)
Update PO catalogs from POT

Keyword Arguments **languages** (*list*) – List of languages to process. Default is `None` so languages are taken from `LANGUAGES` settings.

Returns List of language identifiers for updated catalogs.

Return type list

Project starter

This component eases starting a new project without to recreate basic structure each time.

Project structure and configuration are created using project template. A project template contains some sources to copy in new project directory and render some scripts templates (like settings or page views files).

Once done, the project is ready to be used.

Project template can be either an embedded one from Optimus or an external one available as a Python package, its Python path will be used to reach it.

class optimus.start_project.**ProjectStarter**(*dry_run=False*)
Bases: `object`

Object to create a new project with its settings, directory structure, scripts and assets.

Keyword Arguments **dry_run** (*bool*) – Dry run mode to perform all tasks but never create anything on File System.

dry_run
bool – Dry run mode state.

logger
logging.Logger – Application logger.

check_destination(*basedir*, *name*)
Merge basedir and name into destination path then check if it does not allready exist.

Parameters

- **basedir** (*str*) – Path to directory where to create new project.
- **name** (*str*) – Directory name to create inside `basedir`.

Raises optimus.exception.DestinationExists – If destination allready exists.

Returns Destination path.

Return type string

deploy_assets(*manifest*, *template_fspath*, *destination*)
Copy directories defined in FILES_TO_SYNC from template manifest into created project.

Parameters

- **manifest** (*object*) – Template manifest object.
- **template_fspath** (*str*) – Template path where to get the locale directory.
- **destination** (*str*) – Destination path (the created project directory).

Returns List of deployed asset directories.

Return type list

deploy_language_files(*manifest*, *template_fspath*, *destination*)
Write provided scripts from template into create project

Parameters

- **manifest** (*object*) – Template manifest object.
- **template_fspath** (*str*) – Template path where to get the locale directory.
- **destination** (*str*) – Destination path (the created project directory).

deploy_scripts(*manifest*, *source_path*, *destination*, *context={}*)
Write provided scripts from project template into created project.

Parameters

- **manifest** (*object*) – Template manifest object.
- **source_path** (*str*) – Path to directory containing script sources.
- **destination** (*str*) – Destination path (the created project directory).

Keyword Arguments **context** (*dict*) – Context of variables to give to script template to render. Default to a empty dict.

Returns List of deployed asset directories.

Return type list

get_template_module(*path*)
Return template module if valid.

Parameters **path** (*str*) – Python path to template module.

Raises optimus.exception.TemplateImportError – If template module import fails.

Returns Template module.

Return type string

get_template_pythonpath(*name*)
Return Python path for template.

Parameters **name** (*str*) – Either a full Python path to a template module or an alias defined from optimus.samples.TEMPLATE_ALIAS.

Returns Template module Python path.

Return type string

install (*basedir*, *name*, *template_pythonpath*)
Install new project structure and content from project template.

Parameters

- **basedir** (*str*) – Path to the directory where to create new project.
- **name** (*str*) – Name of the new project, will be also the dir name of the created project, this must be a valid module name (without spaces, special chars, etc..)
- **template_pythonpath** (*str*) – Python path or alias name to the template module.

Returns Path where the new project has been created.

Return type string

render_script (*filepath*, *destination*, *context={}*)
Render script source into created project using `string.Template`.

Parameters

- **filepath** (*str*) – Filepath source.
- **destination** (*str*) – Filepath destination.

Keyword Arguments **context** (*dict*) – Context of variables to give to script template to render. Default to a empty dict.

Python Module Index

o

Index

Symbols

A

B

C

D

E

G

H

I

J

L

M

O

P

R

S

T

U

V