cookiecutter-djangocms3-buildout Documentation

Release 0.6.6

Emencia

May 16, 2015

Contents

CHAPTER 1

Introduction

This is a Cookiecutter template to make a buildout structure to build a Django project using Django CMS, Django Blog Zinnia, Django CKEditor, Django Filebrowser, and many other apps. All of them are allready configured and ready to work. Project integration is made with Compass + Foundation.

Emencia uses this tool for web projects along with our techniques and procedures.

Its goal is to automatically create and initialize a complete project structure so you don't lose time assembling all parts and components.

CHAPTER 2

Table of contents

2.1 Install

There is no need to install anything but Cookiecutter, when it's done you can use any Cookiecutter template directly from a repository (local, hosted on github, etc..).

Warning: This template require a feature only available since `cookiecutter==1.1.0` that is not released yet.
The given PIP requirements file in this repository will help you to install the right version until the release is published.
So currently, you will have to install Cookiecutter using the given `requirements.txt` file in this cookiecutter template repository:

```
pip install -r https://raw.githubusercontent.com/emencia/cookiecutter-djangocms3-buildout/master/re
```

2.1.1 Created projects requirements

Note: If previously you allready have installed Epaster, you don't have to worry about this.

Although the template itself don't need anything but Cookiecutter, a project created with this template will requires some libraries to build it and use it:

- virtualenv;
- Python development library (commonly known as `python-devel`);
- `libjpeg`;
- `zlib`;
- `libfreetype`;

The method to install them depends on your plateform :

- On **Linux** systems you will install them with your package system like `apt-get`, see Pillow documentation for Linux;
- On **MacOSX**, the recommended way is to use `brew` utility, see Pillow documentation for OSX;
- **Windows** system is not supported, you probably can install needed stuff but with some works on your own;

Also if you need to use a **PostgreSQL** database instead of the default **sqlite3** database, you will need a library to build **psycopg2**, this library is called `libpq`.

Note: All created projects are shipped with a README file that contain all necessary details to build it and use it. This will be a simplified procedure with a **Makefile** command to launch the buildout process.

2.2 Usage

This is quite simple, just invoke the Cookiecutter template to create a new project:

```
cookiecutter https://github.com/emencia/cookiecutter-djangocms3-buildout
```

You will be prompted to anwser to some inputs about your project:

project_name [Default:*Project name*]

Project name, should be unique into your repository host;

repo_name [Default:A slug created from project name]

This will be used as the repository name;

repo_username [Default:*emencia*]

Your username to use to push the repository;

repo_host [Default:*github.com*]

The hostname of the repository host;

secret_key [Default:A random key]

The secret key to use in Django settings, you should let the default value;

Then you will be prompted to define the application to enable within your project:

enable_accounts [Default:*no*]

Enable the accounts component;

enable_contact_form [Default:*yes*]

Enable a basic contact form with an optional captcha field;

enable_porticus [Default:*yes*]

Enable Porticus application;

enable_slideshows [Default:*yes*]

Enable Slideshows application;

enable_socialaggregator [Default:*no*]

Enable Social agregator application;

enable_zinnia [Default:*yes*]

Enable Zinnia application;

enable_multiple_languages [Default:*no*]

Enable CMS internationalization;

Take care that question about applications require a strict "yes" value to enable them, all other value are considered as a "no".

Note: In future, Cookiecutter will probably implement a better system to have real questions, but for now their labels are just keywords.

2.2.1 Created projects usage

Once a project has been created, you need to build it to use it. The process is simple. Do it in your project directory:

```
make install
```

When it's finished, active the virtual environment:

```
source bin/active
```

You can then use the project on the development server:

```
django-instance runserver 0.0.0.0:8001
```

Note: `0.0.0.0` is some sort of alias that mean "bind this server on my ip", so if your local ip is "192.168.0.42", the server will be reachable in your browser with the url `http://192.168.0.42:8001/`.

Note: Note the `:8001` that mean "bind the server on this port", this is a required part when you specify an IP. Commonly you can't bind on the port 80 so allways prefer to use a port starting from *8001*.

Note: If you don't know your local IP, you can use `127.0.0.1` that is an internal alias to mean "my own network card", but this IP cannot be reached from other computers (because they have also this alias linked to their own network card).

The first required action is the creation of a CMS page for the home page and also you should fill-in the site name and its domain under `Administration > Sites > Sites > Add site`.

2.3 Projects structure

Projects are created with the many components that are available for use. These components are called **mods** and these mods are already installed and ready to use. You can enable or disable them, as needed.

2.3.1 django-instance

This is the command installed to replace the old `manage.py` script in Django. `django-instance` is located into the `project/bin` directory and is aware of installed eggs from buildout.

2.3.2 Git ignore

Project embeds many `.gitignore` files to avoid to commit some files into your repository.

Principle is to never commit files created from buildout (installed packages, app development sources, etc..), compiled static files, project media files and database.

2.3.3 Makefile

Project embeds a `Makefile` file that contains some usefull commands to build your project.

- `install` to proceed to a new install of this project. Use clean command before if you want to reset a current install;
- `clean` to clean your local repository from all stuff created by buildout and instance usage;
- `delpyc` to remove all `*.pyc` files, this is recursive from the current directory;
- `assets` to minify all assets and collect static files;
- `scss` to compile all SCSS stuffs with Compass;

2.3.4 Foundation

Default SCSS are made for Foundation and all templates are integrated using Foundation components. A complete Foundation 5 install is embedded into the project to work from the first time.

2.3.5 Adding application

If you plan to integrate a new app into a project, always use the buildout system to ensure its portability (pip requirements file is not used in our system).

To do this, just open and edit the `buildout.cfg` file to add the new egg name to be installed. For more details, read the buildout documentation.

Also it is always preferable to use the mods system to configure new added apps and keep the `settings.py` only for Django owned settings.

Note: Project contains a `version.cfg` file that define exact version to use for listed eggs in `buildout.cfg`. All existing mods have their eggs defined in this file, if you need to enable a mod that you skipped during project creation, just find its egg name in `version.cfg` and add it to the eggs in `buildout.cfg`.

2.3.6 How the Mods work

The advantage of centralizing app configurations in their mods is to safely gather together them distinctly from the Django basic settings (like SMTP config, database access, middlewares, etc..). Furthermore it is easier to enable or disable apps.

To create a new mods:

- Create a directory in `project/mods_available/` that contains at least one empty `__init__.py` and a `settings.py` to enable the app (using `settings.INSTALLED_APPS`) in the project and potentially its settings and urls;
- The `settings.py` and `urls.py` files in this directory will be executed automatically by the project (the system loads them after the project ones so that a mods can overwrite the project initial settings and urls);
- N.B. With the Django `runserver` command, a change to these files does not reload the project instance; you need to relaunch it yourself manually;

To enable a new mods, you need to create its symbolic link (**a relative path** to the available mod) in `project/mods_enabled`. To disable it, simply delete the symbolic link.

2.3.7 Available mods

accounts

Enable Django registration and everything you need to allow users to request registration and to connect/disconnect. The views and forms are added so this part can be used.

It includes:

- A view for the login and one for the logout;
- All the views for the registration request (request, confirmation, etc.);
- A view to ask for the reinitialization of a password.

In the `skeleton.html` template, a partial HTML code is commented. Uncomment it to display the *logout* button when the user is connected.

The registration process consists in sending an email (to be configured in the settings) with the registration request to an administrator responsible for accepting them (or not). Once validated, an email is sent to the user to confirm his registration by way of a link. Once this step has been completed, the user can connect.

Also, note that this app use a dummy profile model linked to User object. This profile is dummy because it implement fields for sample but you may not need all of them or you can even may not need about a Profile model, the User object could be enough for your needs. So before to use the syncdb, be sure to watch for the model to change it, then apply your changes to `forms.RegistrationFormAccounts`, `views.RegistrationView` and eventually templates.

admin_style

Enable djangocms-admin-style to enhance the administration interface. Also enable django-admin-shortcuts.

admin-style better fit with DjangoCMS than admin_tools.

Warning: This mod cannot live with admin_tools, you have to choose only one of them.

admin_tools

Enable django-admin-tools to enhance the administration interface. This enables three widgets to customize certain elements and link to filebrowser module (that should allready be enabled).

Warning: This mod cannot live with admin_style, you have to choose only one of them.

assets

Enable django-assets to combine and minify your *assets* (CSS, JS). The minification library used, *yuicompressor*, requires the installation of Java (the OpenJDK installed by default on most Linux systems is sufficient).

In general, this component is required. If you do not intend to use it, you will need to modify the project's default templates to remove all of its occurrences.

Assets are defined in `project/assets.py` and some apps can defined their own `asset.py` file but our main file does not use them.

Our `asset.py` file is divised in three parts :

- BASE BUNDLES: Only for app bundle like Foundation Javascript files or RoyalSlider files;
- MAIN AVAILABLE BUNDLES: Where you defined main bundles for the frontend, use app bundles previously defined;
- ENABLE NEEDED BUNDLE: Bundle you effectively want to use. Bundle that are not defined here will not be reachable from templates and won't be compiled;

ckeditor

Enable and define customization for the CKEditor editor. It is enabled by default and used by Django CKEditor in the cms mod, and also in zinnia.

Note that DjangoCMS use it's own app named "djangocms_text_ckeditor", a djangocms plugin to use CKEditor (4.x).

But Zinnia (and some other generic app) use "django_ckeditor" that ship the same ckeditor but without cms addons.

This mod contains configuration for all of them.

And some useful patches/fixes :

- the codemirror plugin that is missing from the ckeditor's django apps;
- A system to use the "template" plugin (see views.EditorTemplatesListView for more usage details);
- Some overriding to have content preview and editor more near to Foundation;

cms

Django CMS allows for the creation and management of the content pages that constitute your site's tree structure. By default, this component enables the use of filebrowser, Django CKEditor and emencia-cms-snippet (a clone of the snippets' plugin with a few improvements).

By default it is configured to use only one language. See its `urls.py` to find out how to enable the management of multiple languages.

codemirror

Enable Django Codemirror to apply the editor with syntax highlighting in your forms (or other content).

It is used by the snippet's CMS plugin.

contact_form

A simple contact form that is more of a standard template than a full-blown application. You can modify it according to your requirements in its `apps/contact_form/` directory. Its HTML rendering is managed by crispy_forms based on a customized layout.

By default, it uses the recaptcha mods.

crispy_forms

Enable the use of django-crispy-forms and crispy-forms-foundation.

crispy_forms is used to manage the HTML rendering of the forms in a finer and easier fashion than with the simple Django form API.

crispy-forms-foundation is a supplement to implement the rendering with the structure (tags, styles, etc.) used in Foundation.

debug_toolbar

Add django-debug-toolbar to your project to insert a tab on all of your project's HTML pages, which will allow you to track the information on each page, such as the template generation path, the query arguments received, the number of SQL queries submitted, etc.

This component can only be used in a development or integration environment and is always disabled during production.

Note that its use extends the response time of your pages and can provokes some bugs (see the warning at end) so for the time being, this mods is disabled. Enable it locally for your needs but never commit its enabled mod and remember trying to disable it when you have a strange bug.

Warning: Never enable this mod before the first database install or a syncdb, else it will result in errors about some table that don't exist (like "django_site").

emencia_utils

Group together some common and various utilities from `project.utils`.

filebrowser

Add Django Filebrowser to your project so you can use a centralized interface to manage the uploaded files to be used with other components (cms, zinnia, etc.).

The version used is a special version called *no grappelli* that can be used outside of the *django-grapelli* environment.

Filebrowser manage files with a nice interface to centralize them and also manage image resizing versions (original, small, medium, etc..), you can edit these versions or add new ones in the settings.

Note: Don't try to use other resizing app like sorl-thumbnails or easy-thumbnails, they will not work with Image fields managed with Filebrowser.

filer

Mod for django-filer and its DjangoCMS plugin

Only enable it for specific usage because this can painful to manage files with Filebrowser and django-filer enabled in the same project.

flatpages

Enable the use of Django flatpages app in your project. Once it has been enabled, go to the `urls.py` in this mod to configure the *map* of the urls to be used.

google_tools

Add django-google-tools to your project to manage the tags for *Google Analytics* and *Google Site Verification* from the site administration location.

Note: The project is filled with a custom template `project/templates/googletools/analytics_code.html` to use Google Universal Analytics, remove it to return to the old Google Analytics.

logentry

Enable django-logentry-admin to browse all admin log entries at contrary to default Django admin behavior that only display the last entries.

pdb

Add Django PDB to your project for more precise debugging with breakpoints.

N.B. Neither `django_pdb` nor `pdb` are installed by buildout. You must install them manually, for example with pip, in your development environment so you do not disrupt the installation of projects being integrated or in production. You must also add the required breakpoints yourself.

See the the django-pdb Readme for more usage details.

Note: Make sure to put django_pdb after any conflicting apps in INSTALLED_APPS so that they have priority.

So with the automatic loading system for the mods, you should enable it with a name like "zpdb", to ensure that it is loaded at the end of the loading loop.

porticus

Add Django Porticus to your project to manage file galleries.

There is a DjangoCMS plugin for Porticus, it is not enabled by default, you will have to uncomment it in the mod settings.

recaptcha

Enable the Django reCaptcha module to integrate a field of the *captcha* type via the Service reCaptcha. This integration uses a special template and CSS to make it *responsive*.

Note: If you do in fact use this module, go to its mods setting file (or that of your environment) to fill in the public key and the private key to be used to transmit the data required.

By default, these keys are filled in with a *fake* value and the captcha's form field therefore sends back a silent error (a message is inserted into the form without creating a Python *Exception*).

sendfile

Enable django-sendfile that is somewhat like a helper around the **X-SENDFILE headers**, a technic to process some requests before let them pass to the webserver.

Commonly used to check for permissions rights to download some private files before let the webserver to process the request. So the webapp can execute some code on a request without to carry the file to download (than could be a big issue with some very big files).

django-sendfile dependancy in the buildout config is commented by default, so first you will need to uncomment its line to install it, before enabling the mod. Then you will need to create the directory to store the protected medias, because if you store them in the common media directory, they will public to everyone.

This directory must be in the project directory, then its name can defined in the `PROTECTED_MEDIAS_DIRNAME` mod setting, default is to use `protected_medias` and so you should create the `project/protected_medias` directory.

Your webserver need to support this technic, no matter on a recent nginx as it is allready embeded in, on Apache you will need to install the Apache module XSendfile (it should be availabe on your distribution packages) and enable it in the virtualhost config (or the global one if you want), see the Apache module documentation for more details. Then remember to update your virtualhost config with the needed directive, use the Apache config file builded from buildout.

The nginx config template allready embed a rule to manage `project/protected_medias` with sendfile, but it is commented by default, so you will need to uncomment it before to launch buildout again to build the nginx config file.

Note: By default, the mod use the django-sendfile's backend for development that is named `sendfile.backends.development`. For production, you will need to use the right backend for your webserver (like `sendfile.backends.nginx`).

Finally you will need to implement it in your code as this will require a custom view to download the file, see the django-sendfile documentation for details about this. But this is almost easy, you just need to use the `sendfile.sendfile` method to return the right Response within your view.

site_metas

Enable a module in `settings.TEMPLATE_CONTEXT_PROCESSORS` to show a few variables linked to Django sites app in the context of the project views template.

Common context available variables are:

- `SITE.name`: Current *Site* entry name;
- `SITE.domain`: Current *Site* entry domain;
- `SITE.web_url`: The Current *Site* entry domain prefixed with the http protocol like `http://mydomain.com`. If HTTPS is enabled 'https' will be used instead of 'http';

Some projects can change this to add some other variables, you can see for them in `project.utils.context_processors.get_site_metas`.

sitemap

This mod use the Django's Sitemap framework to publish the `sitemap.xml` for various apps. The default config contains ressources for DjangoCMS, Zinnia, staticpages, contact form and Porticus but only ressource for DjangoCMS is enabled.

Uncomment ressources or add new app ressources for your needs (see the Django documentation for more details).

slideshows

Enable the emencia-django-slideshows app to manage slide animations (slider, carousel, etc.). This was initially provided for *Foundation Orbit* and *Royal Slider*, but can be used with other libraries if needed.

socialaggregator

Enable the emencia-django-socialaggregator app to manage social contents.

Note: This app require to fill some API keys settings (like for Twitter API, Facebook API, etc..) to work correctly.

staticpages

This mod uses emencia-django-staticpages to use static pages with a direct to template process, it replace the deprecated mod *prototype*.

thumbnails

Mod for easy-thumbnails a library to help for making thumbnails on the fly (or not).

Generally **this is not recommended**, because by default we allready enable Filebrowser that allready ships a thumbnail system.

urlsmap

django-urls-map is a tiny Django app to embed a simple management command that will display the url map of your project.

zinnia

Django Blog Zinnia allows for the management of a blog in your project. It is well integrated into the cms component but can also be used independently.

2.4 Common topics around project usage

Additionally to Django a created project is based on many other tools you should know, here are their topics.

2.4.1 Compass

Compass is a **Ruby** tool used to compile SCSS sources in **CSS**.

By default, a Django project has its SCSS sources in the `compass/scss/` directory. The CSS Foundation framework is used as the database.

A recent install of Ruby and Compass is required first for this purpose (see RVM if your system installation is not up to date).

Once installed, you can then compile the sources on demand. Simply go to the `compass/` directory and launch this command:

```
compass compile
```

When you are working uninterruptedly on the sources, you can simply launch the following command:

```
compass watch
```

Compass will monitor the directory of sources and recompile the modified sources automatically.

By default the `compass/config.rb` configuration file (the equivalent of *settings.py*‘ in Django) is used. If needed, you can create another one and specify it to Compass in its command (for more details, see the documentation).

RVM

rvm is somewhat like what virtualenv is to Python: a virtual environment.

Difference is that it’s intended for parallel installations of different **Ruby** versions without mixing gems (the **Ruby** application packages) from all Ruby version. In our scenario, it allows you to install a recent version of **Ruby** without affecting your system installation.

This is not required, just an usefull tip to know when developing on old systems with outdated packages.

2.4.2 Webfonts

Often, we use webfonts to display icons instead of images, because a webfont is more flexible to use (it can take any size without to re-upload it) and more light on file size. It is also more *CSS friendly*.

Commonly we use icomoon that is a service to pack a selected set of webfonts to a ZIP archive that you can use to easily embed it in your project.

The first thing is to go on icomoon, create a webfont project and select the needed item from fonts. Then you have a webfont project, you have to download it as a ZIP archive and open it when it’s done.

When you open the archive, you should something like that :

```
icomoon/
-- demo-files
|   -- demo.css
|   -- demo.js
-- demo.html
-- fonts
|   -- icomoon.eot
|   -- icomoon.svg
|   -- icomoon.ttf
|   -- icomoon.woff
-- Read Me.txt
-- selection.json
-- style.css
```

What we need here is the `fonts` directory because it contains the font we need to put in our project assets, and the `style.css` file that contain the icons class name *map*.

So for a created project, first you will copy the fonts directory in `project/webapp_statics` into your project, there should allready be a `fonts` directory with a default dummy font that is not really used, you can safely overwrite it.

Now open the `style.css` from the archive, it should look like this :

```
@font-face {
        font-family: 'icomoon';
        src:url('fonts/icomoon.eot?n45w4u');
        src:url('fonts/icomoon.eot?#iefixn45w4u') format('embedded-opentype'),
                url('fonts/icomoon.woff?n45w4u') format('woff'),
                url('fonts/icomoon.ttf?n45w4u') format('truetype'),
                url('fonts/icomoon.svg?n45w4u#icomoon') format('svg');
        font-weight: normal;
        font-style: normal;
}
[class^="icon-"], [class*=" icon-"] {
        font-family: 'icomoon';
        speak: none;
        font-style: normal;
        font-weight: normal;
        font-variant: normal;
        text-transform: none;
        line-height: 1;

        /* Better Font Rendering =========== */
        -webkit-font-smoothing: antialiased;
        -moz-osx-font-smoothing: grayscale;
}


.icon-left:before {
        content: "\e622";
}
.icon-right:before {
        content: "\e623";
}
.icon-play:before {
        content: "\e62b";
}
```

Not that there are two parts, the first with `@font-face` and `[class^="icon-"], [class*=" icon-"]`, and the second part with some icon class names. Don't mind about the first part, we allready define it in our SCSS component, just copy the whole second part with all class names for your icons.

Then you will have to fill the class names used in the SCSS components `compass/scss/components/_icomoon.scss` in your project, search for this pattern at the end of the file :

```
// Icon list
/*
*
* HERE GOES THE ICONS FROM THE style.css bundled in the icomoon archive
*
*/
```

And put the pasted icon class names after this pattern.

Finally in `compass/scss/app.scss` search for the line containing `@import "components/icomoon";` and uncomment it, now you can compile your SCSS and the webfont icons will be available from your `app.css` file.

2.4.3 Assets management

Why

In the past, assets management was painful with some projects, because their includes was often divided in many different templates. This causing issues like to update some library or retrieve effective code that was working on some template by inherit.

Also, this often results in pages loading dozen of asset files and sometime much more. This is a really bad behavior because it slows page loading and add useless performance charge on the web server.

This is why we use an **asset manager** called django-assets which is a subproject of webassets. Firstly read the webassets documentation to understand how is working its **Bundle** system. Then you can read the django-assets that is only related about Django usage with the settings, templatetags, etc..

How it works

Asset managers generally perform two tasks :

- Regroup some kind of files together, like regrouping all Javascript files in an unique file;
- Minimize the file weight with removing useless white spaces to have the code on unique line;

Some asset manager implement this with their own file processor, some other like webassets are just "glue" between the files and another dedicated *compiler* like yuicompressor.

Environments

Asset management is really useful within integration or production environments and so when developing, the manager is generally disabled and the files are never compiled, you can verify this with looking at your page's source code.

make assets

Project have a `make assets` command that is useful **on integration and production environment** to deploy and update your assets in the `static/` directory. In fact **this command is always required in these environments** when you deploy a new update. Also you should never use it on development environment because it can cause you many troubles.

What does this command :

1. Collecting all static files from your project and installed apps to your `settings.STATIC_ROOT` directory;
2. Use django-assets to *compile* all defined bundles using previously collected files;
3. Re-collecting static files again to collect the compiled bundle files;

Static files directories

In your `settings.py` file you should see :

```
STATIC_ROOT = join(PROJECT_PATH, 'static')
```

It define the *front* static file directory. But **never put yourself a file in this directory**, it is **reserved** for collected files in **integration and production environment** only.

All static files sources will go in the `project/webapp_statics` directory, it is defined in the *assets* mod:

```
ASSETS_ROOT = join(PROJECT_PATH, 'webapp_statics/')
STATICFILES_DIRS += (ASSETS_ROOT,)
```

This way we allways have separated directories for the sources and the compiled files. This is required to never commit compiled files and avoid conflict between development and production.

The rule

Never, ever, put CSS stylesheets in your templates, NEVER. You can forget it, this will go in production and forgeted for a long time, this can be painful for other developers that coming after you. So **always add CSS stylesheets by the way of SCSS sources** using Compass.

For Javascript code this is different, sometime we need to generate some code using Django templates for some specific cases. But if you use a same Javascript code in more than one template (using inheriting or so), you must move the code to a Javascript file.

Developers should never have to search in templates to change some CSS or Javascript code that is used in more than one page.

2.4.4 Developing application

Sometimes, you will need to develop some new app package or improve them without to embed them within the project.

You have two choices to do that:

- Use `develop` buildout variable to simply add your app to the developped apps, your app have to exists at the root of buildout project;
- Use `vcs-extend-develop` buildout variable to define a repository URL to the package sources;

Even they have the same base name *develop*, these two ways are differents:

- The first one simply add a symbolic link to the package in your Python install without to manage it as an installed eggs, it will be accessible as a Python module installed in the Python virtual environment. This method does not require that your app have a repository or have been published on PyPi;
- The second one install the targeted package from a given repository instead of a downloaded package from PyPi, it act like an installed eggs but from which you can edit the source and publish to the repository. And so your app name have to be defined in the buildout's egg variable, buildout will see it in `vcs-extend-develop` and will not try to install it from PyPi but from the given repository url;

In all ways, your apps is allways a full package structure that mean this is not a simple Python module, but its package structure containing stuff like `README` file and `setup.py` at the base of the directory then the Python module containing the code. Trying to use a simple Python module as a develop app will not work.

Which one to use and when

- If you want to **develop a new package**, it's often much faster to create its package directory structure at the root of your buildout project then use it within `develop`. You would move it to `vcs-extend-develop` when you have published it;
- If you want to **develop an allready published package**, you will use `vcs-extend-develop` with its repository url, this so you will be able to edit it, commit changes then publish it;

Most of Emencia's apps are allready setted within `vcs-extend-develop` in the buildout config for development environment (`development.cfg`) but disabled, just uncomment the needed one.

Take care, an Egg that is installed from a repository url is validated on its version number if defined in the `versions.cfg`, and so if your develop egg contains a version number less than the one defined in `versions.cfg`, buildout will try to get the most recent version from PyPi, so allways manage the app version number.

2.4.5 PO-Projects

It aims to ease PO translations management between developpers and translation managers.

The PO-Projects client is pre-configured in all created projects but disabled by default. When enabled, its config file is automatically generated (in `po_projects.cfg`), don't edit this file because it will be regenerated each time buildout is used.

The principe is that **developpers and translators does not have anymore to directly exchange PO files**. The developpers update the PO to the translation project on PO-Project webservice, translators update translations on PO-Project service frontend and developpers can get updated PO from the webservice.

To use it, you will have first to enable it in the buildout config, to install the client package, fill the webservice access and buildout part. Then when it's done, you have to create a project on PO-Project webservice using its frontend, then each required language for translation using the same locale names that the ones defined in the project settings.

There is only two available actions from the client :

Push action The `push` action role is to send updated PO (from Django extracts) from the project to the PO-Project webservice.

Technically, the client will archive the locale directory into a tarball then send it to the webservice, that will use it to update its stored PO for each defined locales.

Common way is (from the root of your project):

```
cd project
django-instance makemessages -a
cd ..
po_projects push
```

Pull action The `pull` action role is to get the updated translations from the webservice and install into the project.

Technically, the client will download a tarball of the latest locale translations from the webservice and deploy it to your project, note that it will totally overwrite the project's locale directory.

Common way is (from the root of your project):

```
po_projects pull
```

Then reload your webserver.

Note that the client does not manage your repository, each time you change your PO files (from Django `makemessages` action or `pull` client action) you still have to commit them.

2.4.6 Gestus

The Gestus client is pre-configured in all created projects, its config file is automatically generated (in `gestus.cfg`), don't edit it because it will be regenerated each time buildout is used.

You can register your environment with the following command :

```
gestus register
```

Remember this should only be used in integration or production environment and you will have to fill a correct accounts in the `EXTRANET` part.

2.4.7 Dr Dump

Dr Dump is an utility to help you to dump and load datas from your Django project's apps. It does not have any command line interface, just a buildout recipe (emencia-recipe-drdump) that will generate some bash scripts (`datadump` and `dataload`) in your `bin` directory so you can use them directly to dump your data into a `dumps` directory.

If the recipe is enabled in your buildout config (this is the default behavior), its bash scripts will be generated again each time you invoke a buildout.

Buildout will probably remove your dumps directory each time it re-install Dr Dump and Dr Dump itself will overwrite your dumped data files each time you invoke it dump script. So remember backup your dumps before doing this.

Note that Dr Dump can only manage app that it allready know in the used map, if you have some other packaged app or project's app that is not defined in the map you want to use, you have two choices :

- Ask to a repository manager of Dr Dump to add your apps, for some *exotic* or uncommon apps it will probably be refused;
- Download the map from the repository, embed it in your buildout project and give its path into the `dependancies_map` recipe variable so it will use it.

The second one is the most easy and flexible, but you will have to manage yourself the map to keep it up-to-date with the original one.

2.5 History

Versions come from git tags, not package version because, err.. this is not a Python package.

2.5.1 Version 0.6.6 - 2015/05/16

- Enforce `django-tagging==0.3.4` (to avoid a bug with django<=1.7);
- Review and update `assets.py`, close #10;
- Some assets cleanup, close #9;
  - Added missing default images for *Royal Slider*;
  - Removed Foundation3 Javascript stuff;
  - Cleaning main frontend script `app.js`;
  - Added MegaMenu stuff;
- Big update on `contact_form` app:
  - Fix print message on template;
  - Reorganise admin view;
  - Use `django-import-export` for exporting contact datas;
  - Don't print captcha on form when `settings.DEBUG` is `True`;

2.5.2 Version 0.6.5 - 2015/05/03

- Cleaning documentations;
- Restored doc stuff to automatically build mod documentations;
- Updated to `django-cms==3.0.13`;
- Enforce `django-contrib-comments==1.5.0` (to avoid a bug with django<=1.7);
- Integrated `django-logentry-admin` as a default enabled mod, close #8;
- Fixed doc config to get the right version number from git tags;

2.5.3 Version 0.6.1 - 2015/04/20

- Added cookiecutter context in `project/__init__.py` file;

2.5.4 Version 0.6.0 - 2015/04/19

- Better documentation;

2.5.5 Version 0.5.0 - 2015/04/17

- Enabled cms translation and some settings from cookiecutter context, close #4;

2.5.6 Version 0.4.0 - 2015/04/16

- Removed unused variables in `cookiecutter.json`;
- Changed ignored files from jinja to target some files to use as templates;
- Changed template for `skeleton.html` to remove occurences to not enabled apps;
- Added cookiecutter context usage to remove unused sitemap parts, close #5;
- Changed buildout.cfg to be more flexible without some enabled apps;

2.5.7 Version 0.3.0 - 2015/04/15

- Added Git repo initialization in the post generation hook;
- Added a message at the end of the post generation hook to display some help;
- Changed some variables from `cookiecutter.json` for repository infos;

2.5.8 Version 0.2.0 - 2015/04/13

- Added post generation hook to enable mods after install;
- Use cookiecutter context to remove eggs in `buildout.cfg` egg list;

2.5.9 Version 0.1.0 - 2015/04/12

- First version started from emencia_paste_djangocms_3 structure version `1.4.0`;
- Not ready to be used yet, it misses some things for now;