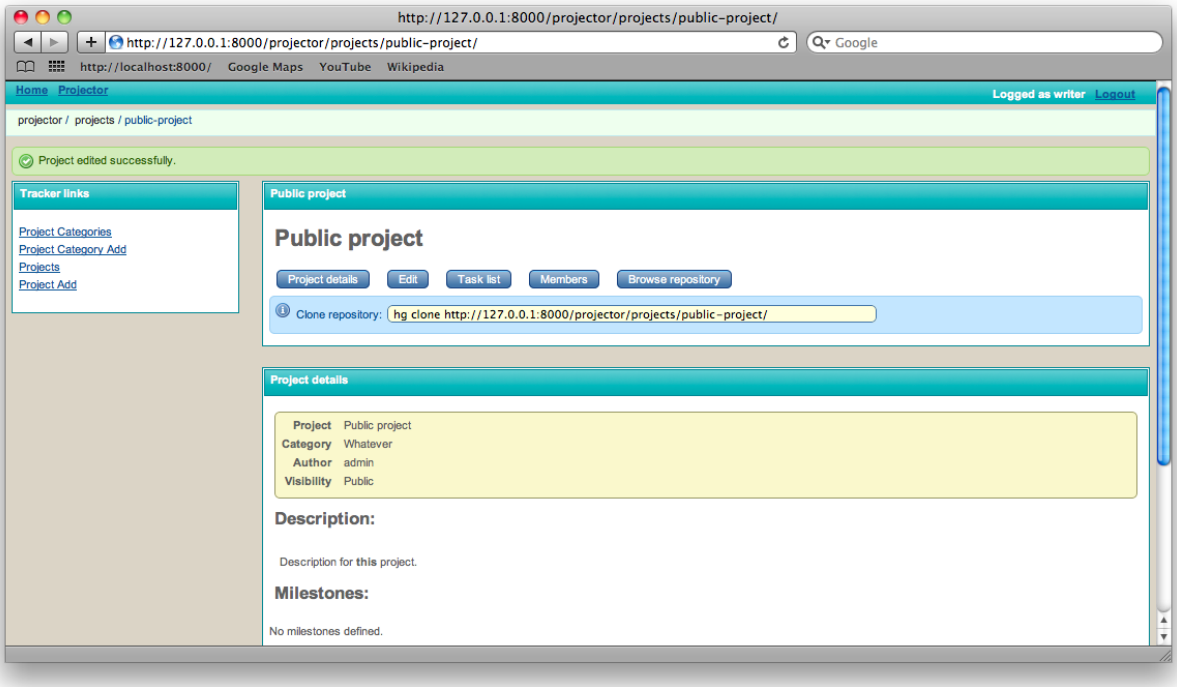
django-projector Documentation

Release 0.1.9

Lukasz Balcerzak

March 19, 2014

`django-projector` is a project management application with task tracker and repository backend integration. Aimed to work with Django 1.2 or later. We are sick of Trac [1] and wanted to create simple application which can be easily customized or plugged into existing systems.



[1] Don't get us wrong, Trac is great tool but we believe that django's pluggable applications are far easier to configure and deploy.

CHAPTER 1

Features

- Mercurial repository integration
- Easy repositories forking
- Granual permissions management (see *Authorization and permissions*)
- Scalable architecture (AMQP) build on top of excellent celery
- Task tracker with full history of changes
- Repository web browser
- Customizable workflow for each project
- Grouping tasks in milestones
- Roadmap
- Teams support
- Documents based on restructuredText
- Email notification
- Make use of django-richtemplates so templates are ready to use out of the box

CHAPTER 2

Incoming

Here are some additional points which are the target for future release.

- Wiki per project
- Plugin system
- Code review
- Sphinx integration
- Other version control systems in backend (git, subversion...)
- Charts, statistics, graphs, plots, analyzies
- Functional timeline
- django-piston integration for RESTful API

Warning: This application is at early-development stage but we strongly encourage you to give it a try if you are looking for project management toolkit for your Django based project. Still, it probably should **NOT** be used in production as it wasn't fully tested and may contain security issues.

CHAPTER 3

Source code

Source code is along with issue tracker is available at http://bitbucket.org/lukaszb/django-projector/.

CHAPTER 4

Demo project

Demo project have been deployed at https://forge.django-projector.org. It is still rather experimental.

CHAPTER 5

License

`django-projector` is released under MIT license. You should get a copy of the license with source distribution, at the root location, within `LICENSE` file.

CHAPTER 6

Documentation

Installation:

6.1 Installation

`django-projector` is aimed to work with Django 1.2 or later. Moreover, we strongly suggest to use virtualenv and virtualenvwrapper - great tools for creating temporary environment to work on.

6.1.1 Requirements

Requirements should be installed along with projector itself by the setuptools. There is also `requirements.txt` file at the root of source distribution with all needed packages. To install dependencies one should run pip [1] command:

```
pip install -r requirements.txt
```

6.1.2 Trying it out

`django-projector` comes with boundled example project. It can be easily used - refer to *this document* for more details.

6.2 Configuration

After you hook `django-projector` into your project (see *Installation*) you should probably change some of configuration variables.

6.2.1 Available settings

Those configurable variables should be defined at the project's settings module, just like standard Django's variables.

[1] pip is tool similar to easy_install with some more power (like smooth integration with virtualenv, `freeze` command, package uninstallation, search and others.

PROJECTOR_ALWAYS_SEND_MAILS_TO_MEMBERS

Default: `False`

If set to `True`, any change to project would send an email to each projects' members regardless of their individual preferences.

PROJECTOR_BANNED_PROJECT_NAMES

Default:

```
(
    'account', 'accounts',
    'add',
    'admin', 'admins',
    'api',
    'author', 'authors',
    'ban',
    'category', 'categories',
    'change',
    'create',
    'default',
    'delete',
    'edit', 'edits',
    'etc',
    'issue', 'issues',
    'mail', 'mails',
    'message', 'messages',
    'manager', 'managers',
    'private',
    'profile', 'profiles',
    'projects',
    'register', 'registration',
    'remove',
    'task', 'tasks',
    'update',
    'user', 'users',
    'view',
)
```

List of names which are restricted during project creation.

Note: By specifying own list, we in fact extend default list. We mention this as most of the settings are overridable - this one is not.

PROJECTOR_BASIC_AUTH_REALM

Default: `'Projector Basic Auth'`

Text which would appear during basic authorization process within projector's context. Projects' owners can override this *per project*.

PROJECTOR_CHANGESETS_PAGINATE_BY

Default: `10`

Number of changesets listed at one page.

PROJECTOR_CREATE_PROJECT_ASYNCHRONOUSLY

Default: `True`

When new project is created some actions are made using Djangos' signals. By default those actions are made asynchronousely by new thread in order not to block client.

PROJECTOR_DEFAULT_PROJECT_WORKFLOW

Default: `projector.conf.default_workflow`

Path to object defining default workflow for new projects.

Object must define following iterables: *components*, *task_types*, *priorities* and *statuses*. Each one should contain dictionaries with following key/value pairs:

- **components**: *name*
- **task_types**: *name*, *order*
- **priorities**: *name*, *order*
- **statuses**: *name*, *oder*, *is_resolved*, *is_initial*

See source code pointed by default value for more detail.

PROJECTOR_DEFAULT_VCS_BACKEND

Default: `'hg'`

One of *aliases* specified at `PROJECTOR_ENABLED_VCS_BACKENDS`. See vcs's documentation for available aliases.

PROJECTOR_EDITABLE_PERMISSIONS

Default:

```
(
    'change_project',
    'change_config_project',
    'view_project',
    'can_read_repository',
    'can_write_to_repository',
    'can_change_description',
    'can_change_category',
    'can_add_task',
    'can_change_task',
    'can_delete_task',
    'can_view_tasks',
    'can_add_member',
    'can_change_member',
    'can_delete_member',
    'can_add_team',
    'can_change_team',
    'can_delete_team',
)
```

List of permission codenames allowed to be edited by projects' owners.

Note: Removing variables from this tuple (by setting own with subset of available permissions) would not affect permissions - it only tells projector to show forms for permission editing with specified codenames.

PROJECTOR_ENABLED_VCS_BACKENDS

Default: `['hg', 'git']`

Iterable of vcs *aliases*. To check what backends are available run:

```
>>> import vcs
>>> vcs.backends.BACKENDS.keys()
['hg', 'git']
```

See more at vcs's documentation.

PROJECTOR_FORK_EXTERNAL_ENABLED

Default: `False`

If set to `True` users would be allowed to fork projects from external locations (read more at *External fork*).

Warning: We **DO NOT** take any responsibility caused by using external forking. Reason is simple - some users could use this functionality to attack external hosts by sending crafted values to the fork form. This should be validated by the form first, though.

PROJECTOR_FORK_EXTERNAL_MAP

Default:

```
{
    'bitbucket.org': 'projector.forks.bitbucket.BitbucketForkForm',
    'github.com': 'projector.forks.github.GithubForkForm',
}
```

Dictionary of forms to be used for external forking. Keys would be used as choices at the first step of external forking process. Values should be paths to the fork form. Read more at *External fork*.

PROJECTOR_FROM_EMAIL_ADDRESS

Default: would try to get value from `settings.DEFAULT_FROM_EMAIL`.

Email address used as sender for all mails send by projector.

PROJECTOR_HG_PUSH_SSL

Default: `False`.

If set to `True`, underlying mercurial engine would transmit data using encryption. This setting has precedence over vcs specific configuration.

PROJECTOR_HIDDEN_EMAIL_SUBSTITUTION

Default: `'HIDDEN_EMAIL'`.

Used as default substitution for hidden emails while using `projector.templatetags.hide_email()` filter (if no parameter is specified).

PROJECTOR_MAX_PROJECTS_PER_USER

Default: `50`

Specifies maximum number of projects one user may create.

PROJECTOR_MILESTONE_DEADLINE_DELTA

Default: `60` (60 days)

This is default value of time delta (in days) added to current date during milestone creation.

PROJECTOR_MILIS_BETWEEN_PROJECT_CREATION

Default: `15000` (15 seconds)

After user created a project, he/she need to wait for time specified with this setting until another project may be created by him/her.

PROJECTOR_PRIVATE_ONLY

Default: `False`

If `True` then only *private* projects may be created. Does *not* affect existing projects.

PROJECTOR_PROJECTS_ROOT_DIR

Default: `None`

If not specified, no repositories would be created. Must be valid directory path.

PROJECTOR_PROJECTS_HOMEDIR_GETTER

Default: `projector.utils.helpers.get_homedir`

Location of the function which should return relative project path. In order to calculate full path of homedir, `projector.models.Project` calls pointed function and appends result to the `PROJECTOR_PROJECTS_ROOT_DIR` value.

It is possible to change this location and override function. It takes one required `project` parameter - instance of `projector.models.Project`. Default implementation returns simply stringified primary key of the given `project`.

PROJECTOR_TASK_EMAIL_SUBJECT_SUMMARY_FORMAT

Default:

```
"[$project] #$id: $summary"
```

This is default subject format for messages related with tasks. Allows to move name placeholders (`$project`, `$id`, `$summary`). All placeholders are optional - but adviced, obviousely.

6.2.2 `get_config_value(key)`

6.3 Authorization and permissions

This project management system aims to be used by small to middle companies.

6.3.1 Object-level permissions

`django-projector` make use of django-guardian to handle object-level permissions. Check out it's documentation to see detailed information on the topic.

6.3.2 Project permissions

Following permissions are defined for each project:

- `change_project`
- `view_project`
- `can_read_repository`
- `can_write_to_repository`
- `can_change_description`
- `can_change_category`
- `can_add_task`
- `can_change_task`
- `can_delete_task`
- `can_view_tasks`
- `can_add_member`
- `can_change_member`
- `can_delete_member`
- `can_add_team`
- `can_change_team`
- `can_delete_team`

Usage:

6.4 Example projector-based project

New in version 0.1.7.

We have included example project for `django-projector` and it can be found at (surprise!) `example_project` directory within both repository and source release. As `django-projector` has many dependancies (see *Installation*) it may be hard to start at the first glance but this sample project can be used as entry point.

Note: We use example project to run *tests*.

In order to run example project we need to make some preparation first.

6.4.1 Prepare

Step 1 - media files

When specifing commands we assume we are at `example_project` directory. Before we can run example project we need to include media files. Thanks to django-richtemplates this is as easy as running one management command:

```
python manage.py import_media richtemplates projector
```

This would fetch all necessary media files and put them into `MEDIA_ROOT` defined at settings module.

Step 2 - database

Now we need to create database (we use `sqlite3` backend for sample project). Type following command:

```
python manage.py syncdb
```

Step 3 - fire up a worker

We need to run a celery worker for some heavy jobs to be done asynchronously. As we use celery we need to start it at one terminal:

```
python manage.py celeryd -l DEBUG
```

Step 4 - finalize

In fact there is no step 4 - simply run development server:

```
python manage.py runserver
```

... and open `http://localhost:8000` location in a browser.

6.5 Teamwork

`django-projector` is all about making life easier for project's members. Thus, we try to implement the best practices in this area, and new proposals are always welcome!

6.5.1 Mappings

Most of the times within Django project we use `django.contrib.auth` application to store and manage users (`auth.User`) and groups (`auth.Group`).

Note: Note that we refer to model using standard notation `applabel.ModelName` which is used around django community. We prefer that over typing full python path to the specific class.

Those are very simply models and give developers ability to wrap them around they own classes. We use them too and this section describes how we do it at `django-projector`.

Most important model within application is `Project`. We connect it with users by `Membership` model. Moreover, we also use `Team` - it provides connection between `Project`, `auth.User` and `auth.Group` models.

6.5.2 Membership

Each user is related with project by `Membership` instance.

Administration

Membership for the author of the project is created automatically, and is given all available permissions and thus it's member is referenced as project's administrator.

Project's adminstrator can add new member, change permissions of existing ones or remove members if necessary. Project owner's permissions cannot be changed.

Admin can also manage *teams*.

Other members

User may be project's *member* without associated `Membership` instance if he or she is member of a `auth.Group` (by the instance of `Team` - see *below* for more detail).

AnonymousUser member

Since Django 1.2 authentication backends may support anonymous user and django-guardian implements this functionality (see more at it's documentation). As so, it is possible to add `auth.AnonymousUser` as a project's member and manage it's permissions as with any other user.

Warning: It is possible to give out administration privileges to anonymous user this way. Some views (like task creation or project edition) requires user to be logged in but project's owner should be careful about anonymous user's permission management.

Convert to `Team`

Any user may be converted into `Team` instance. Well, this is not totally true - in fact, by conversion to `Team` we mean *set a team flag* on the user's profile. Conversion is available if user profile's `is_team` attribute is False and there is no `auth.Group` instance named same as the user.

Conversion is done within user's dashboard and each step of conversion is described below:

1. User clicks on *Convert to Team* button at his or her dashboard.

2. If there is `auth.Group` named as the user, `ValidationError` is raised.
3. User confirms conversion.
4. `auth.Group` instance named same as the user is created. This group is automatically added to `User.groups`.
5. `UserProfile.is_team` attribute is set to `True`. From now on, accessing `UserProfile.group` would return `auth.Group` instance created in previous step.

Conversion's api is provided by `Team` manager's method `projector.managers.TeamManager.convert_from_user()`.

6.5.3 Team

Any `auth.Group` may be used to create `Team` instance which bounds `auth.Group` and `Project`. Normally, one would create group using *account conversion*.

One user may be member of many teams. Single project may be managed by many users *and* many teams. It may be confusing but it's really simple.

6.6 Projects documentation

6.6.1 The Basics

Return to *Projects documentation*.

Creating projects

As `Project` is most important (for `django-projector`) and rather non-trival model we would like to present whole process of creating projects, step by step.

Let's create our first project:

```
>>> from django.contrib.auth.models import User
>>> from projector.models import Project
>>> joe = User.objects.get(username='joe')
>>> project = Project.objects.create(author=joe, name='foobar')
```

Now we need to add *metadata* for created project. This is done in a few steps:

- *set_memberships*
- *set_author_permissions*
- *create_workflow*
- *create_config*
- *create_repository*

set_memberships

Now we need to add membership for the author and if he/she is a *team* we would create a `Team` instance binding `Project` and `auth.Group`. We simply call `set_memberships` method which would do this for us:

```
>>> project.members.all()
[]
>>> project.set_memberships()
>>> project.members.all()
[<User: joe>]
```

set_author_permissions

After author become first member of the project he/she still cannot, i.e. *change* the project:

```
>>> joe.has_perm('projector.change_project', project)
>>> False
```

Note: At projector's views if requested user is author of the project, permissions are not checked at all. This is intenional, as less database hits is always better. On the other hand, if i.e. user would give project away to other user, he still should have all permissions - at least until new owner wouldn't took them from original author.

We can now set permissions:

```
>>> project.set_author_permissions()
>>> joe.has_perm('projector.change_project', project)
>>> True
```

create_workflow

Workflow is a set of statuses, components etc. for each project. Default set of objects is pointed by `PROJECTOR_DEFAULT_PROJECT_WORKFLOW`. Workflow itself may be modified for each project. We may pass a string pointing to the python object or an object itself. Again, simply fire up the method:

```
>>> project.create_workflow()
```

create_config

Per project configuration is available at `Config`. This model defines all *changable* settings for each project all projects need one:

```
>>> project.create_config()
```

create_repository

If `PROJECTOR_CREATE_REPOSITORIES` is set to `True` then we should create repository for the project:

```
>>> from projector.settings import get_config_value
>>> if get_config_value('CREATE_REPOSITORIES'):
        project.create_repository()
```

setup

`Project` comes with `setup` method which would call all preparation methods at given instance. Is is possible to pass `vcs_alias` and `workflow` parameters but they are not required. So all of the above code may be called with little less effort:

```
>>> from django.contrib.auth.models import User
>>> from projector.models import Project
>>> joe = User.objects.get(username='joe')
>>> project = Project.objects.create(author=joe, name='foobar')
>>> project.setup()
```

Using manager

At previous section, *Creating projects*, we have seen that there are some methods which should be called every time new `Project` is created. We can call *setup* method to make the process less tedious. On the other hand it may be even better if we can simply save `Project` instance into database and call `setup` method asyncronously (if `CREATE_PROJECT_ASYNCHRONOUSLY` is set to `True`).

`Project.objects.create_project`

There is a special signal `setup_project` which is called by the `ProjectManager`'s `create_project` method. It is preferred way to create new `Project`:

```
>>> from django.contrib.auth.models import User
>>> from projector.models import Project
>>> joe = User.objects.get(username='joe')
>>> project = Project.objects.create_project(author=joe, name='foobar')
```

We can also specify `vcs_alias` or `workflow` parameters directly:

```
>>> project = Project.objects.create_project(author=joe, name='foobar', vcs_alias='hg', workflow=None
```

6.6.2 Forking

Return to *Projects documentation*.

New in version 0.1.8.

In many situations we need to *clone* one project, make some changes (*progress*), review them and eventually merge them into upstream. Or we know that our changes won't be accepted (or shouldn't be) but we still need to make them for our own reasons. When we talk about software repository we would call that clone a *branch*. But if we refer to project as a whole, we often call this projects' copies *forks*.

`django-projector` allows user to fork another project not only within it's own context. It is also possible to fork a project from external location, i.e. from Bitbucket.

Internal fork

This is standard fork functionality which may be found at other forges. Procedure is simple:

1. Jack decides to fork project named *joe's project*
2. At the *joe's project* main page Jack clicks on *Fork* button
3. Jack is redirected to his new forked project named as original one

External fork

At his or her dasboard, user can find a *Fork project* button. This does **not** refer to *internal fork*. External forking only allows to fork from external location.

To enable this functionality, it's necessary to set `PROJECTOR_FORK_EXTERNAL_ENABLED = True` at settings file. Moreover, `PROJECTOR_FORK_EXTERNAL_MAP` dict setting should be set properly (see *Configuration*).

Warning: We **DO NOT** take any responsibility caused by using external forking. Reason is simple - some users could use this functionality to attack external hosts by sending crafted values to the fork form. Values should be validated by the form first, though.

How to write external fork form

External fork form should subclass `projector.forks.base.BaseExternalForkForm` and implement `fork` method:

- `fork()`: this method should implement action required to create `projector.models.Project` instance. Note that real fork procedure is fired by project creation handler. We may create a project in whatever way we want here but most basic scenario is to pass `author`, `name` and `public` attributes to the constructor of `projector.models.Project` class.

  Note: All exceptions at `fork` method should be caught and eventually propagated but with type `ProjectorError` (or a subclass of it). It is necessary for `projector.forms.ExternalForkWizard` to properly notify user if any error has occured during forking process. Those are not validation errors as `fork` method should be called only after form is cleaned.

Moreover, `projector.forks.base.BaseExternalForkForm` subclasses need to be initialized with `django.http.HttpRequest` as first positional argument (this is required for further form's validation).

Base fork form class comes with one field `as_private`. After form validation it is possible to check if project should be forked as *public* or *private* by calling `is_public` form's method. This method would return `True` or `False`.

After form is implemented we can hook it at the `PROJECTOR_FORK_EXTERNAL_MAP`.

We advice to review code of `projector.forks.bitbucket` module to see full example.

Development

6.7 Testing

We decided that best way to test `django-projector` would be to use not complicated, boundled *example project*. It means less maintainance as tests have to be make within django context anyway. On the other hand, this also implies that tests are run in specifically defined environment (django settings module) so it may not always fit into real project. This is still little disadvantage and less maintainance means less work before release.

6.7.1 How to test

In order to run test suite we simply run:

```
python setup.py test
```

This should invoke preparation process and fire up Django test runner.

Note: It is also possible to run test suite using management command but please remember that we have to use some custom settings and therefor it is **required** to use `example_project/settings_test.py`. Simply run following command within example project:

```
$ python manage.py test projector --settings=settings_test
```

API

6.8 API Reference

6.8.1 Core

Return to *API Reference*.

Controllers

Return to *Core*. This module extends standard Django function views and allows us to use *so called* class-based-views. Django itself contains many features to power up class based approach to the topic (for instance `django.utils.decorators.method_decorator`).

During development of `django-projector` we just copied codes much too often and class-based views allow us to complete many tasks in a much simpler way.

View

class `projector.core.controllers.`**`View`**(*request*, **args*, ***kwargs*)

Main view class. Implemntation is focused around `__call__` method and *classmethod* `new`.

Subclasses should implement `response` method but it is not required.

Class-level attributes:

Attribute login_required Default: `False`. If set to `True`, `response` method would be wrapped with standard `django.contrib.auth.decorators.login_required`.

Attribute template_name Default: `'base.html'` If `response` method returns a `dict` then this value is used to render context.

Instance-available attributes:

Attribute self.context Context dictionary. During initialization it is set to empty dict (`{}`).

Attribute self.request `django.http.HttpRequest` instance passed in by the url resolver.

Attribute self.args Positional parameters passed in by url resolver.

Attribute self.kwargs Named parameters passed in by url resolver.

Implementation example:

```
from projector.core.controllers import View
from projector.models import Project

class ProjectList(View):
```

```
    login_required = True
    template_name = 'project_list.html'

    def response(self, request):
        self.context = Project.objects.all()
        return self.context

class ProjectDetail(View):

    login_required = True
    template_name = 'project_detail.html'

    def response(self, request, project_id):
        self.context['project'] = get_object_or_404(Project, id=project_id)
        return self.context
```

Typically, we would hook such defined views at `urls.py` module:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('projector.views',
    (r'^projects/$', 'ProjectList'),
    (r'^projects/(?P<project_id>\d+)/$', 'ProjectDetail'),
)
```

Note that url resolvers pass all parameters to the given functions. Our subclasses of `View` class are treated very similar to functions as they should return `HttpResponse` object after being called.

__after__()
: Method called at the final step of the view-class processing. If `django.http.HttpResponse` instance is returned it is propagated to the view handler.

Note: Adding i.e. `context` at this stage is pointless - response should have been already prepared and changing `context` wouldn't change the response. `__after__` method should be implemented for other purposes, like checking status code of prepared response. Response object is available by accessing `self._response`. This attribute is available **only** at this stage.

__before__()
: Method called just after class is created and all passed parameters are set on the class, so it is possible to access those attributes.

  If this method would return `django.http.HttpResponse` instance, it would be propagated.

__call__()
: Should be overridden only for special cases as it runs `__after__` method after `response` method is executed. By subclassing it is possible to change the order of method calls.

response(*request*, **args*, ***kwargs*)
: Should be overridden at subclass. It always requires `django.http.HttpRequest` instance to be passed as first positional argument.

  May return a `dict` or `django.http.HttpResponse` instance. If `dict` is returned it is treated as *context* and view would try to render it using `self.template_name`.

  Returns `dict` or `django.http.HttpResponse` instance

Exceptions

Exceptions raised by `django-projector`.

All internal error classes should extend from one of those exceptions.

`ProjectorError` should always be *top-level* exception class.

Return to *Core*.

ProjectorError

class `projector.core.exceptions.`**`ProjectorError`**
Main `django-projector` exception.

ConfigAlreadyExist

class `projector.core.exceptions.`**`ConfigAlreadyExist`**

ForkError

class `projector.core.exceptions.`**`ForkError`**

6.8.2 Forms

Return to *API Reference*.

ProjectCreateForm

ProjectEditForm

ProjectForkForm

ConfigForm

UserProfileForm

ExternalForkWizard

UserConvertToTeamForm

DashboardAddMemberForm

6.8.3 Managers

Return to *API Reference*.

ProjectManager

6.8.4 Models

Return to *API Reference*.

Project

See also `ProjectManager`.

Config

Milestone

Component

Priority

Status

Transition

TaskType

AbstractTask

TaskRevision

Task

Membership

Team

UserProfile

6.8.5 Utils

Utilities and helpers for `django-projector`.

Return to *API Reference*.

Basic

`projector.utils.abspath`

Returns absolute path for given `*paths`.

projector.utils.str2obj

projector.utils.**str2obj**(*text*)

Returns object pointed by the string. In example:

```
>>> from projector.models import Project
>>> point = 'projector.models.Project'
>>> obj = str2obj(point)
>>> obj is Project
True
```

projector.utils.using_projector_profile

projector.utils.**using_projector_profile**()

Returns True if AUTH_PROFILE_MODULE is set to projector.UserProfile and False otherwise.

Email

projector.utils.email.**extract_emails**(*text*)

Returns list of emails founded in a given text.

Helpers

projector.utils.helpers.get_homedir

projector.utils.helpers.**get_homedir**(*project*)

Returns unique home directory path for each project. Returned path should be relative to PROJECTOR_PROJECTS_ROOT_DIR.

Parameters **project** – instance of projector.models.Project

---

Note: This is simplest possible implementation but on the other hand returned value depends solely on the primary key of the given project. It is important **not** to use other fields as they may change. Primary key should never be changed while even creation date (created_at field) may be edited.

For tests however, this is not acceptable as test runner rollbacks queries which should normally be committed. Rollbacks make ID value not stable - as we run more and more tests we always got projects with same id (starting with 1 for each test). That said, test runner must use other *project homedir generator*.

Implementation:

```
return str(project.pk)
```

6.8.6 Signals

Return to *API Reference*.

Provided signals

django-projector provides following signals:

post_fork

After fork is done `post_fork` signal should be send.

`providing_args`:

Name	Description
`fork`	Should be a `Project` instance forked from another project

setup_project

Should be send by `Project` instance.

Name	Description
`instance`	`Project` instance which should be setup
`vcs_alias`	If given, would be used as backend for new repository
`workflow`	If given, would override default workflow

6.8.7 Views

Return to *API Reference*.

Project views

Return to *Views*.

ProjectView

ProjectDetailView

ProjectListView

ProjectCreateView

ProjectForkView

Project repository views

Return to *Views*.

RepositoryView

RepositoryBrowse

RepositoryFileDiff

RepositoryFileRaw

RepositoryFileAnnotate

RepositoryChangesetList

RepositoryChangesetDetail

Users views

Return to *Views*.

UserListView

UserHomepageView

UserProfileDetailView

UserDashboardView

UserDashboardForkView

UserDashboardConvert2TeamView

UserDashboardAddMember

CHAPTER 7

Other topics

- *genindex*
- *search*

p