
Django Groups Manager Documentation

Release 1.0.0

Vittorio Zamboni

Jan 29, 2022

Contents

1	Documentation	3
1.1	Installation	3
1.2	Basic usage	4
1.3	Django auth models integration	4
1.4	Settings	5
1.4.1	Auth model synchronization	5
1.4.2	Permissions	5
1.4.3	Templates	6
1.4.4	Slugify function	6
1.4.5	Defaults	6
1.5	Use cases	7
1.5.1	Projects management with Proxy Models	7
1.5.2	Projects management with Model Mixins	9
1.5.3	Model mixins example	9
1.5.4	Resource assignment via role permissions	11
1.5.5	Resource assignment via group type permissions	12
1.5.6	Custom member model	13
1.5.7	Custom signals	14
1.5.8	Expiring memberships	15
1.6	API	16
1.6.1	Models	16
1.6.2	Perms	16
1.7	Templates	16
1.7.1	Requirements	16
1.7.2	Structure	16
1.7.3	Style	17
1.8	Tests	17
1.9	TODO	17
1.10	Changelog	17
2	Changelog	19
3	Indices and tables	21

Django Groups Manager allows to manage groups based on [django-mptt](#).

The application offers three main classes: *Group*, *Member* and *GroupMember*. It's possible to *map* groups and members with Django's auth models, in order to use external applications such [django-guardian](#) to handle permissions.

The basic idea of Groups is that each *Group* instance could have a *Group* instance as parent (this relation is managed via [django-mptt](#)).

The code is hosted on [github](#).

1.1 Installation

Requirements

- Python ≥ 3.5
- Django ≥ 2

First of all, install the latest build with `pip`:

```
pip install django-groups-manager
# and, in case you want per-object permissions related features
pip install django-guardian
```

or the repository master version for latest updates:

```
pip install https://github.com/vittoriozamboni/django-groups-manager/archive/master.
↪ zip
```

Add `groups_manager` to installed apps:

```
INSTALLED_APPS += (
    # 'guardian', # add as well to use permissions related features
    'groups_manager',
)
```

Run `syncdb` or `migrate`:

```
python manage.py migrate
```

If you want to use standard templates, add `groups_manager`'s urls from `urls.py`:

```
urlpatterns = ('',
    # ...
```

(continues on next page)

(continued from previous page)

```
url(r'^groups-manager/', include('groups_manager.urls', namespace='groups_manager
↪')),
    # ...
)
```

Supported templates are based on bootstrap3, and django-bootstrap3 application is required:

```
pip install django-bootstrap3
```

If you don't want to use bootstrap3, you can override forms (see Templates documentation).

1.2 Basic usage

A simple use case for this application is the tracking of customers groups. Each *organization* can have more than one *division*, and a *member* can be in more than one:

```
from groups_manager.models import Group, GroupType, Member

# Create group types (optional)
organization = GroupType.objects.create(label='Organization')
division = GroupType.objects.create(label='Division')

# Organization A has 'commercials' and 'managers'
org_a = Group.objects.create(name='Org A, Inc.', group_type=organization)
org_a_commercials = Group.objects.create(name='Commercials', group_type=division, ↪
↪parent=org_a)
org_a_managers = Group.objects.create(name='Managers', group_type=division, ↪
↪parent=org_a)
# Tina is a commercial
tina = Member.objects.create(first_name='Tina', last_name='Rossi')
org_a_commercials.add_member(tina)
# Jack is a manager
jack = Member.objects.create(first_name='Jack', last_name='Black')
org_a_managers.add_member(jack)

#Assign objects to members or groups
product = Product.objects.create(name='Fancy product') # Product has 'sell_product' ↪
↪permission
tina.assign_object(org_a_commercials, product)
tina.has_perm('sell_product', product) # True
jack.has_perm('sell_product', product) # False
budget = Budget.objects.create(name='Facilities', amount=5000) # has 'use_budget' ↪
↪permission
org_a_managers.assign_object(budget)
tina.has_perm('use_budget', budget) # False
jack.has_perm('use_budget', budget) # True
```

1.3 Django auth models integration

It is possible to auto-map Group and Member instances with django.contrib.auth.models Group and User. To enable mapping, "AUTH_MODELS_SYNC" setting must be set to True (default: False), and also Group and Member instances attribute django_auth_sync (that is True by default).

Add to your settings file:

```
GROUPS_MANAGER = {
    'AUTH_MODELS_SYNC': True,
}
```

This will generate auth's groups and users each time a new groups_manager's group or member is created. In addition, every time a groups_manager's GroupMember instance is generated (either via instance creation or via Group's add_member method), the django user is added to django group.

1.4 Settings

The application can be configured via settings with GROUPS_MANAGER dictionary. Valid keys and values are described below.

1.4.1 Auth model synchronization

- "AUTH_MODELS_SYNC": enables Group, Member and GroupMember synchronization with django's Group and User (default: False);
- "AUTH_MODELS_GET_OR_CREATE": use get_or_create method instead of create for django's Group and User models when associating to Group or Member (default: True);
- "GROUP_NAME_PREFIX": prefix used for autogenerated django Group's name (default: "DGN_")
- "GROUP_NAME_SUFFIX": suffix used for autogenerated django Group's name. The special value "\$random" can be used for generate a pseudo-unique suffix of length 8 (the first block of an UUID4) (default: "\$random")
- "USER_USERNAME_PREFIX": prefix used for autogenerated django User's username (default: "DGN_")
- "USER_USERNAME_SUFFIX": suffix used for autogenerated django User's username. The special value "\$random" can be used (default: "\$random")

1.4.2 Permissions

- **"PERMISSIONS" dictionary: this setting controls the assign_object method of a GroupMember instance.** Each key controls a specific group type. Values are lists (or, in case of "owner", also a dictionary) with a combination of permissions' prefixes "view" (view), "change" (change), "delete" (delete) characters. Obviously, a "view_modelname" permission must be added to the model permissions. You can also add your custom permissions in form of <prefix> where your permission is <prefix>_modelname.

Valid keys are:

- "owner": a list or a dictionary (with keys as roles' codename attribute). This object-permissions are assigned directly to the user (default: ['view', 'change', 'delete'])
- "group": a string. This object-permissions are assigned to the related group (default: ['view', 'change'])
- "groups_upstream": a string. This object-permissions are assigned to the ancestors groups (default: ['view'])
- "groups_downstream": a string. This object-permissions are assigned to the descendants groups (default: [])

- "groups_siblings": a string. This object-permissions are assigned to the siblings groups (default: ['view'])

Note: The four special permission names "add", "view", "change", and "delete" are translated to `<permission>_<model_name>` string during permission's name lookup. This allows to use a standard permission policy (*view*, *change*, *delete*) but also allows to use *custom permissions*.

An example of permissions assigned by role can be found on use cases.

1.4.3 Templates

- `TEMPLATE_STYLE`: name of the templates folder inside "groups_manager". By default is "bootstrap3", this means that templates are searched inside folder "groups_manager/bootstrap3"

1.4.4 Slugify function

- `SLUGIFY_FUNCTION`: function used to slugify codenames;
- `SLUGIFY_USERNAME_FUNCTION`: function used to slugify auto-created usernames.

Note: Prior to *1.1.0*, the default function was *awesome-slugify*. Due to its licence, it cannot be distributed with *django-groups-manager*.

By default Django's slugify is used, with conversion to lower and, only for username, `_` as separator.

To use a different function with it's own parameters, you can define a lambda function in settings (i.e. using *awesome-slugify*):

```
from slugify import slugify

GROUPS_MANAGER = {
    'SLUGIFY_FUNCTION': lambda s: slugify(s, to_lower=True),
    'SLUGIFY_USERNAME_FUNCTION': lambda s: slugify(s, to_lower=True, separator="_")
}
```

1.4.5 Defaults

Default values are:

```
GROUPS_MANAGER = {
    # User and Groups sync settings
    'AUTH_MODELS_SYNC': False,
    'GROUP_NAME_PREFIX': 'DGM_',
    'GROUP_NAME_SUFFIX': '_$$random',
    'USER_USERNAME_PREFIX': 'DGM_',
    'USER_USERNAME_SUFFIX': '_$$random',
    # Permissions
    'PERMISSIONS': {
        'owner': ['view', 'change', 'delete'],
        'group': ['view', 'change'],
        'groups_upstream': ['view'],
```

(continues on next page)

(continued from previous page)

```

        'groups_downstream': [],
        'groups_siblings': ['view'],
    },
    # Templates
    'TEMPLATE_STYLE': "bootstrap3",
    'SLUGIFY_FUNCTION': lambda s: slugify(s).lower(),
    'SLUGIFY_USERNAME_FUNCTION': lambda s: slugify(s).lower().replace('-', '_'),
}

```

1.5 Use cases

1.5.1 Projects management with Proxy Models

John Boss is the project leader. Marcus Worker and Julius Backend are the django backend guys; Teresa Html is the front-end developer and Jack College is the student that has to learn to write good backends. The Celery pipeline is owned by Marcus, and Jack must see it without intercatations. Teresa can't see the pipeline, but John has full permissions as project leader. As part of the backend group, Julius has the right of viewing and editing, but not to stop (delete) the pipeline.

1) Define models in models.py:

```

from groups_manager.models import Group, GroupType

class Project(Group):
    # objects = ProjectManager()

    class Meta:
        proxy = True

    def save(self, *args, **kwargs):
        if not self.group_type:
            self.group_type = GroupType.objects.get_or_create(label='Project')[0]
        super(Project, self).save(*args, **kwargs)

class WorkGroup(Group):
    # objects = WorkGroupManager()

    class Meta:
        proxy = True

    def save(self, *args, **kwargs):
        if not self.group_type:
            self.group_type = GroupType.objects.get_or_create(label='Workgroup')[0]
        super(WorkGroup, self).save(*args, **kwargs)

class Pipeline(models.Model):
    name = models.CharField(max_length=100)

    class Meta:
        permissions = (('view_pipeline', 'View Pipeline'), )

```

Warning: Remember to define a `view_modelname` permission.

2) Connect creation and deletion signals to the proxy models (*This step is required if you want to sync with django auth models*):

```
from django.db.models.signals import post_save, post_delete
from groups_manager.models import group_save, group_delete

post_save.connect(group_save, sender=Project)
post_delete.connect(group_delete, sender=Project)
post_save.connect(group_save, sender=WorkGroup)
post_delete.connect(group_delete, sender=WorkGroup)
```

3) Creates groups:

```
project_main = testproject_models.Project(name='Workgroups Main Project')
project_main.save()
django_backend = testproject_models.WorkGroup(name='WorkGroup Backend',
    ↳parent=project_main)
django_backend.save()
django_backend_watchers = testproject_models.WorkGroup(name='Backend Watchers',
    parent=django_backend)
django_backend_watchers.save()
django_frontend = testproject_models.WorkGroup(name='WorkGroup FrontEnd',
    ↳parent=project_main)
django_frontend.save()
```

4) Creates members and assign them to groups:

```
john = models.Member.objects.create(first_name='John', last_name='Boss')
project_main.add_member(john)
marcus = models.Member.objects.create(first_name='Marcus', last_name='Worker')
julius = models.Member.objects.create(first_name='Julius', last_name='Backend')
django_backend.add_member(marcus)
django_backend.add_member(julius)
teresa = models.Member.objects.create(first_name='Teresa', last_name='Html')
django_frontend.add_member(teresa)
jack = models.Member.objects.create(first_name='Jack', last_name='College')
django_backend_watchers.add_member(jack)
```

5) Create the pipeline and assign custom permissions:

```
custom_permissions = {
    'owner': ['view', 'change', 'delete'],
    'group': ['view', 'change'],
    'groups_upstream': ['view', 'change', 'delete'],
    'groups_downstream': ['view'],
    'groups_siblings': [],
}
pipeline = testproject_models.Pipeline.objects.create(name='Test Runner')
marcus.assing_object(django_backend, pipeline, custom_permissions=custom_permissions)
```

Note: The full tested example is available in repository source code, `testproject's tests.py` under `test_proxy_models` method.

1.5.2 Projects management with Model Mixins

Mixins allows to create shared apps based on django-groups-manager. The mixins approach has pros and cons.

Pros:

- models are completely customizable (add all fields you need);
- all fields are in the same table (with subclassed models, only extra fields are stored in the subclass table);
- better for shared applications (the “original” django-groups-manager tables don’t share entries from different models).

Cons:

- all external foreign keys must be declared in the concrete model;
- all signals must be declared with concrete models.

1.5.3 Model mixins example

The following models allow to manage a set of Organizations with related members (from organization app). In this example, a `last_edit_date` is added to models, and member display name has the user email (if defined).

1) Define models in models.py:

```
from groups_manager.models import GroupMixin, MemberMixin, GroupMemberMixin, \
    GroupMemberRoleMixin, \
    GroupEntity, GroupType, \
    group_save, group_delete, member_save, member_delete, group_member_save, group_
    member_delete

class OrganizationMemberRole(GroupMemberRoleMixin):
    pass

class OrganizationGroupMember(GroupMemberMixin):
    group = models.ForeignKey('OrganizationGroup', related_name='group_membership')
    member = models.ForeignKey('OrganizationMember', related_name='group_membership')
    roles = models.ManyToManyField(OrganizationMemberRole, blank=True)

class OrganizationGroup(GroupMixin):
    last_edit_date = models.DateTimeField(auto_now=True, null=True)
    short_name = models.CharField(max_length=50, default='', blank=True)
    country = CountryField(null=True, blank=True)
    city = models.CharField(max_length=200, blank=True, default='')

    group_type = models.ForeignKey(GroupType, null=True, blank=True, on_delete=models.
    SET_NULL,
                                related_name='%(app_label)s_%(class)s_set')
    group_entities = models.ManyToManyField(GroupEntity, null=True, blank=True,
                                related_name='%(app_label)s_%(class)s_set'
    )

    django_group = models.ForeignKey(DjangoGroup, null=True, blank=True, on_
    delete=models.SET_NULL)
    group_members = models.ManyToManyField('OrganizationMember',
    through=OrganizationGroupMember,
```

(continues on next page)

(continued from previous page)

```

through_fields=('group', 'member'),
related_name='% (app_label)s_%(class)s_set')

class Meta:
    permissions = (('manage_organization', 'Manage Organization'),
                   ('view_organization', 'View Organization'))

class GroupsManagerMeta:
    member_model = 'organizations.OrganizationMember'
    group_member_model = 'organizations.OrganizationGroupMember'

def save(self, *args, **kwargs):
    if not self.short_name:
        self.short_name = self.name
    super(OrganizationGroup, self).save(*args, **kwargs)

@property
def members_names(self):
    return [member.full_name for member in self.group_members.all()]

class OrganizationMember(MemberMixin):
    last_edit_date = models.DateTimeField(auto_now=True, null=True)
    django_user = models.ForeignKey(DjangoUser, null=True, blank=True, on_
↪delete=models.SET_NULL,
                                   related_name='% (app_label)s_%(class)s_set')

class GroupsManagerMeta:
    group_model = 'organizations.OrganizationGroup'
    group_member_model = 'organizations.OrganizationGroupMember'

def __unicode__(self):
    if self.email:
        return '%s (%s)' % (self.full_name, self.email)
    return self.full_name

def __str__(self):
    if self.email:
        return '%s (%s)' % (self.full_name, self.email)
    return self.full_name

```

2) Connect creation and deletion signals to the models

(This step is required if you want to sync with django auth models):

```

post_save.connect(group_save, sender=OrganizationGroup)
post_delete.connect(group_delete, sender=OrganizationGroup)

post_save.connect(member_save, sender=OrganizationMember)
post_delete.connect(member_delete, sender=OrganizationMember)

post_save.connect(group_member_save, sender=OrganizationGroupMember)
post_delete.connect(group_member_delete, sender=OrganizationGroupMember)

```

3) Customize the flag for AUTH_MODEL_SYNC

If you plan to create a reusable app and to let users decide if sync or not with Django auth models **independently** from groups_manager settings, you should define a separated function that returns the boolean value from your own

settings:

```
def organization_with_mixin_get_auth_models_sync_func(instance):
    return organization.SETTINGS['DJANGO_AUTH_MODEL_SYNC'] # example

def organization_group_member_save(*args, **kwargs):
    group_member_save(*args, get_auth_models_sync_func=organization_get_auth_models_
↳sync_func, **kwargs)

def organization_group_member_delete(*args, **kwargs):
    group_member_delete(*args, get_auth_models_sync_func=organization_get_auth_models_
↳sync_func, **kwargs)

post_save.connect(organization_group_member_save, sender=OrganizationGroupMember)
post_delete.connect(organization_group_member_delete, sender=OrganizationGroupMember)
```

Note: The full tested example is available in repository source code, testproject's tests.py under test_model_mixins method.

1.5.4 Resource assignment via role permissions

John Money is the commercial referent of the company; Patrick Html is the web developer. The company has only one group, but different roles. John can view and sell the site, and Patrick can view, change and delete the site.

1) Define models in models.py:

```
class Site(Group):
    name = models.CharField(max_length=100)

    class Meta:
        permissions = (('view_site', 'View site'),
                       ('sell_site', 'Sell site'), )
```

2) Create models and relations:

```
from groups_manager.models import Group, GroupMemberRole, Member
from models import Site
# Group
company = Group.objects.create(name='Company')
# Group Member roles
commercial_referent = GroupMemberRole.objects.create(label='Commercial referent')
web_developer = GroupMemberRole.objects.create(label='Web developer')
# Members
john = Member.objects.create(first_name='John', last_name='Money')
patrick = Member.objects.create(first_name='Patrick', last_name='Html')
# Add to company
company.add_member(john, [commercial_referent])
company.add_member(patrick, [web_developer])
# Create the site
site = Site.objects.create(name='Django groups manager website')
```

3) Define custom permissions and assign the site object:

```
custom_permissions = {
    'owner': {'commercial-referent': ['sell_site'],
              'web-developer': ['change', 'delete'],
              'default': ['view']},
    'group': ['view'],
    'groups_upstream': ['view', 'change', 'delete'],
    'groups_downstream': ['view'],
    'groups_siblings': ['view'],
}
john.assign_object(company, site, custom_permissions=custom_permissions)
patrick.assign_object(company, site, custom_permissions=custom_permissions)
```

4) Check permissions:

```
john.has_perms(['view_site', 'sell_site'], site) # True
john.has_perm('change_site', site) # False
patrick.has_perms(['view_site', 'change_site', 'delete_site'], site) # True
patrick.has_perm('sell_site', site) # False
```

Note: The full tested example is available in repository source code, testproject's tests.py under test_roles method.

1.5.5 Resource assignment via group type permissions

Permissions can also be applied to related groups filtered by group types. Instead of simply using a list to specify permissions one can use a dict to specify which group types get which permissions.

Example

John Money is the commercial referent of the company; Patrick Html is the web developer. John and Patrick can view the site, but only Patrick can change and delete it.

1) Define models in models.py:

```
class Site(Group):
    name = models.CharField(max_length=100)

    class Meta:
        permissions = (('view_site', 'View site'),
                       ('sell_site', 'Sell site'), )
```

2) Create models and relations:

```
from groups_manager.models import Group, GroupType, Member
from models import Site

# Parent Group
company = Group.objects.create(name='Company')

# Group Types
developer = GroupType.objects.create(label='developer')
referent = GroupType.objects.create(label='referent')
```

(continues on next page)

(continued from previous page)

```
# Child groups
developers = Group.objects.create(name='Developers', group_type=developer,
    ↳parent=company)
referents = Group.objects.create(name='Referents', group_type=referent,
    ↳parent=company)

# Members
john = Member.objects.create(first_name='John', last_name='Money')
patrick = Member.objects.create(first_name='Patrick', last_name='Html')

# Add to groups
referents.add_member(john)
developers.add_member(patrick)

# Create the site
site = Site.objects.create(name='Django groups manager website')
```

3) Define custom permissions and assign the site object:

```
custom_permissions = {
    'owner': [],
    'group': ['view'],
    'groups_downstream': {'developer': ['change', 'delete'], 'default': ['view']},
}
john.assign_object(company, site, custom_permissions=custom_permissions)
```

4) Check permissions:

```
john.has_perm('view_site', site) # True
john.has_perm('change_site', site) # False
john.has_perm('delete_site', site) # False
patrick.has_perms(['view_site', 'change_site', 'delete_site'], site) # True
```

Note: The full tested example is available in repository source code, testproject's tests.py under test_group_types_permissions method.

1.5.6 Custom member model

By default, Group's attribute members returns a list of Member instances. If you want to create also a custom Member model in addition to custom Group, maybe you want to obtain a list of custom Member model instances with members attribute. This can be obtained with GroupsManagerMeta's member_model attribute. This class must be defined in Group subclass/proxy. The value of the attribute is in <application>.<model_name> form.

1) Define models in models.py:

```
from groups_manager.models import Group, Member

class Organization(Group):

    class GroupsModelMeta:
        model_name = 'myApp.OrganizationMember'
```

(continues on next page)

(continued from previous page)

```
class OrganizationMember(Member):  
    pass
```

2) Call Organization members attribute:

```
org_a = Organization.objects.create(name='Org, Inc.')  
boss = OrganizationMember.objects.create(first_name='John', last_name='Boss')  
org_a.add_member(boss)  
org_members = org_a.members # [<OrganizationMember: John Boss>]
```

Note: A tested example is available in repository source code, testproject's tests.py under test_proxy_model_custom_member and test_subclassed_model_custom_member methods.

1.5.7 Custom signals

If you redefine models via proxy or subclass and you need to manage sync permissions with a different setting (like MY_APP['AUTH_MODELS_SYNC']) you need to use different signals functions when saving objects and relations. Signals functions accept kwargs:

- `get_auth_models_sync_func`: a function that returns a boolean (default honours GROUPS_MANAGER['AUTH_MODELS_SYNC'] setting), that also take an instance parameter to allow additional checks;
- `prefix` and `suffix` on `group_save`': override GROUPS_MANAGER['GROUP_NAME_PREFIX'] and GROUPS_MANAGER['GROUP_NAME_SUFFIX'];
- `prefix` and `suffix` on `member_save`': override GROUPS_MANAGER['USER_USERNAME_PREFIX'] and GROUPS_MANAGER['USER_USERNAME_SUFFIX'];

So, for example, your wrapping functions will be like this:

```
class ProjectGroup(Group):  
  
    class Meta:  
        permissions = (('view_projectgroup', 'View Project Group'), )  
  
    class GroupsManagerMeta:  
        member_model = 'testproject.ProjectGroupMember'  
        group_member_model = 'testproject.ProjectGroupMember'  
  
class ProjectMember(Member):  
  
    class Meta:  
        permissions = (('view_projectmember', 'View Project Member'), )  
  
class ProjectGroupMember(GroupMember):  
    pass  
  
def project_get_auth_models_sync_func(instance):  
    return MY_APP['AUTH_MODELS_SYNC']
```

(continues on next page)

(continued from previous page)

```

def project_group_save(*args, **kwargs):
    group_save(*args, get_auth_models_sync_func=project_get_auth_models_sync_func,
                prefix='PGS_', suffix='_Project', **kwargs)

def project_group_delete(*args, **kwargs):
    group_delete(*args, get_auth_models_sync_func=project_get_auth_models_sync_func,
    ↪ **kwargs)

def project_member_save(*args, **kwargs):
    member_save(*args, get_auth_models_sync_func=project_get_auth_models_sync_func,
                prefix='PMS_', suffix='_Member', **kwargs)

def project_member_delete(*args, **kwargs):
    member_delete(*args, get_auth_models_sync_func=project_get_auth_models_sync_func,
    ↪ **kwargs)

def project_group_member_save(*args, **kwargs):
    group_member_save(*args, get_auth_models_sync_func=project_get_auth_models_sync_
    ↪ func, **kwargs)

def project_group_member_delete(*args, **kwargs):
    group_member_delete(*args, get_auth_models_sync_func=project_get_auth_models_sync_
    ↪ func, **kwargs)

post_save.connect(project_group_save, sender=ProjectGroup)
post_delete.connect(project_group_delete, sender=ProjectGroup)

post_save.connect(project_member_save, sender=ProjectMember)
post_delete.connect(project_member_delete, sender=ProjectMember)

post_save.connect(project_group_member_save, sender=ProjectGroupMember)
post_delete.connect(project_group_member_delete, sender=ProjectGroupMember)

```

Note: A tested example is available in repository source code, testproject's tests.py under test_signals_kwargs method.

1.5.8 Expiring memberships

Members can be added to groups with an optional date that specifies when the membership expires.

expiration_date property is only used to indicate when the membership expires and has no effect on the permissions. How this property is used is up to the user of the library. This can be useful for example to filter out expired memberships or periodically delete them.

Set expiration date to one week from today

```
import datetime
from django.utils import timezone

john = models.Member.objects.create(first_name='John', last_name='Boss')
expiration = timezone.now() + datetime.timedelta(days=7)
project_main.add_member(john, expiration_date=expiration)
```

1.6 API

1.6.1 Models

Main models

Group Models

Relation Management

1.6.2 Perms

perms module has permissions related utilities.

1.7 Templates

1.7.1 Requirements

The supported templates requires bootstrap3. Forms are based on django-bootstrap3 application, that can be installed with:

```
pip install django-bootstrap3
```

This application is used only to render forms. If you don't want to use it's default rendering, you can override the `form_template.html` file as described in the example below.

1.7.2 Structure

Templates are organized in different sections, inside `groups_manager/bootstrap3` folder.:

```
- groups_manager.html (extends "base.html")
| - groups_manager_home.html (menu with links to models lists)
| - <model>.html (model base, i.e. "member", extended by all model templates)
|   - <model>_list.html
|   - <model>_detail.html
|   - <model>_form.html (includes form_template.html)
|   - <model>_confirm_delete.html
```

There are different blocks:

- breadcrumbs: usually displayed on top of the page, with app - model - page links
- sidebar: menu available for the application and <model> actions (add, edit, etc)

- `content`: the main content of the page

To change a template, creates the same structure inside an application loaded after `groups_manager` in your `INSTALLED_APPS`. For example, to change `form_template.html` file, create the folders “`groups_manager/bootstrap3/`” and put the file “`form_template.html`” inside.

1.7.3 Style

By default, style is “`bootstrap3`”: this means that templates are searched inside folder “`groups_manager/bootstrap3`”. To change this behaviour, edit setting `TEMPLATE_STYLE`: it will be used in the views:

```
template_name = 'groups_manager%s/groups_manager.html' % TS
```

1.8 Tests

First of all, you need to clone the repository and create a virtualenv with all dependencies. Then you can run tests through the `manage.py` test command:

```
virtualenv django-groups-manager-test
cd django-groups-manager-test
source bin/activate
git clone https://github.com/vittoriozamboni/django-groups-manager.git
cd django-groups-manager/testproject
pip install -r requirements.txt
python manage.py test testproject groups_manager
```

1.9 TODO

1.10 Changelog

- **22-01-29 (1.2.0):**
 - Support Django 4: uses correct url parser function `re_path` (see issue #60, thank you Lukas Hennies!)
 - Updated `intro.rst`, fixed wrong example (see issue #57, thank you Areski Belaid!)
- **21-04-11 (1.1.0):**
 - Removed `awesome-slugify` from requirements. It needs to be installed separately due to his licence (see issue #54, thank you BoPeng!);
 - Added a new settings to customize the slugify functions for username and other cases.
- **20-06-17 (1.0.2):**
 - Changed `jsonfield2` to `jsonfield` in requirements and tests (see issue #49, thank you ioio!);
- **20-03-07 (1.0.1):**
 - Amended Django 3 deprecations
 - Documentation changes
- **19-12-10 (1.0.0):**

- Dropped support for Django < 2 and Python 2.*
- **19-01-11 (0.6.2):**
 - Added migrations for `expiration_date` and verbose names
- **18-01-18 (0.6.1):**
 - Added support for Django 2
- **17-12-09 (0.6.0) (thank you Oskar Persson!):**
 - Added group type permission handling
 - Added `expiration_date` attribute
 - Added support to `django-jsonfield`
- **16-11-08 (0.5.0):**
 - Added models mixins
 - Removed compatibility for Django < 1.7
- **16-10-10 (0.4.2):**
 - Added initial migration
 - Removed null attributes from m2m relations
- **16-04-19 (0.4.1):**
 - Removed unique to group name (this cause issues when subclassing, since it does not allows to have same names for different models)
 - Fixed issue with python 3 compatibility in `templatetags` (thank you Josh Manning!)
- **16-03-01 (0.4.0):**
 - Added kwargs to signals for override settings parameters
 - Added `remove_member` to group as a method (previously must be done manually)
- **16-02-25 (0.3.0):**
 - Added permissions assignment to groups
 - Added support for Django 1.8 and 1.9
- **15-05-05 (0.2.1):**
 - Added ‘add’ to default permissions
- **15-05-05 (0.2.0):**
 - Changed retrieval of permission’s name: ‘view’, ‘change’ and ‘delete’ will be translated to ‘<name>_<model_name>’, the others are left untouched (see [permission name policy](#))
 - Added `GroupsManagerMeta` class to `Group` that allows to specify the member model to use for members list (see [custom Member model](#))
- 14-10-29 (0.1.0): Initial version

CHAPTER 2

Changelog

- **22-01-29 (1.2.0):**
 - Support Django 4: uses correct url parser function *re_path* (see issue #60, thank you Lukas Hennies!)
 - Updated intro.rst, fixed wrong example (see issue #57, thank you Areski Belaid!)
- **21-04-11 (1.1.0):**
 - Removed *awesome-slugify* from requirements. It needs to be installed separately due to his licence (see issue #54, thank you BoPeng!);
 - Added a new settings to customize the slugify functions for username and other cases.
- **20-06-17 (1.0.2):**
 - Changed jsonfield2 to jsonfield in requirements and tests (see issue #49, thank you ioio!);
- **20-03-07 (1.0.1):**
 - Amended Django 3 deprecations
 - Documentation changes
- **19-12-10 (1.0.0):**
 - Dropped support for Django < 2 and Python 2.*
- **19-01-11 (0.6.2):**
 - Added migrations for expiration_date and verbose names
- **18-01-18 (0.6.1):**
 - Added support for Django 2
- **17-12-09 (0.6.0) (thank you Oskar Persson!):**
 - Added group type permission handling
 - Added *expiration_date* attribute
 - Added support to django-jsonfield

- **16-11-08 (0.5.0):**
 - Added models mixins
 - Removed compatibility for Django < 1.7
- **16-10-10 (0.4.2):**
 - Added initial migration
 - Removed null attributes from m2m relations
- **16-04-19 (0.4.1):**
 - Removed unique to group name (this cause issues when subclassing, since it does not allows to have same names for different models)
 - Fixed issue with python 3 compatibility in templatetags (thank you Josh Manning!)
- **16-03-01 (0.4.0):**
 - Added kwargs to signals for override settings parameters
 - Added `remove_member` to group as a method (previously must be done manually)
- **16-02-25 (0.3.0):**
 - Added permissions assignment to groups
 - Added support for Django 1.8 and 1.9
- **15-05-05 (0.2.1):**
 - Added 'add' to default permissions
- **15-05-05 (0.2.0):**
 - Changed retrieval of permission's name: 'view', 'change' and 'delete' will be translated to '<name>_<model_name>', the others are left untouched (see [permission name policy](#))
 - Added `GroupsManagerMeta` class to `Group` that allows to specify the member model to use for members list (see [custom Member model](#))
- **14-10-29 (0.1.0):** Initial version

CHAPTER 3

Indices and tables

- `genindex`
- `modindex`
- `search`