

---

# **guardian Documentation**

*Release 1.4.6.dev0+ng0e9acdd.d20160810*

**Lukasz Balcerzak**

August 10, 2016



<b>1</b>	<b>Overview</b>	<b>3</b>
1.1	Features . . . . .	3
1.2	Incoming . . . . .	3
1.3	Source and issue tracker . . . . .	3
1.4	Alternate projects . . . . .	3
<b>2</b>	<b>Installation</b>	<b>5</b>
<b>3</b>	<b>Configuration</b>	<b>7</b>
<b>4</b>	<b>Optional settings</b>	<b>9</b>
4.1	GUARDIAN_RAISE_403 . . . . .	9
4.2	GUARDIAN_RENDER_403 . . . . .	9
4.3	GUARDIAN_TEMPLATE_403 . . . . .	9
4.4	ANONYMOUS_USER_NAME . . . . .	9
4.5	GUARDIAN_GET_INIT_ANONYMOUS_USER . . . . .	10
<b>5</b>	<b>User Guide</b>	<b>11</b>
5.1	Example project . . . . .	11
5.2	Assign object permissions . . . . .	11
5.3	Check object permissions . . . . .	14
5.4	Remove object permissions . . . . .	17
5.5	Admin integration . . . . .	17
5.6	Custom User model . . . . .	19
5.7	Performance tuning . . . . .	20
5.8	Caveats . . . . .	22
<b>6</b>	<b>API Reference</b>	<b>25</b>
6.1	Admin . . . . .	25
6.2	Backends . . . . .	26
6.3	Core . . . . .	27
6.4	Decorators . . . . .	27
6.5	Forms . . . . .	29
6.6	Management commands . . . . .	30
6.7	Managers . . . . .	30
6.8	Mixins . . . . .	31
6.9	Models . . . . .	33
6.10	Shortcuts . . . . .	33
6.11	Utilities . . . . .	38

6.12	Template tags . . . . .	39
<b>7</b>	<b>Development</b>	<b>41</b>
7.1	Overview . . . . .	41
7.2	Testing . . . . .	42
7.3	Supported versions . . . . .	43
7.4	Changelog . . . . .	44
<b>8</b>	<b>License</b>	<b>51</b>
<b>9</b>	<b>Indices and tables</b>	<b>53</b>
	<b>Python Module Index</b>	<b>55</b>

**Date** August 10, 2016

**Version** 1.4.6.dev0+ng0e9acdd.d20160810

**Documentation:**



---

## Overview

---

`django-guardian` is an implementation of object permissions for Django providing extra *authentication backend*.

### 1.1 Features

- Object permissions for Django
- AnonymousUser support
- High level API
- Heavily tested
- Django's admin integration
- Decorators

### 1.2 Incoming

- Admin templates for `grappelli`

### 1.3 Source and issue tracker

Sources are available at [issue-tracker](#). You may also file a bug there.

### 1.4 Alternate projects

Django 1.2 still has *only* foundation for object permissions<sup>1</sup> and `django-guardian` make use of new facilities and it is based on them. There are some other pluggable applications which does *NOT* require latest 1.2 version of Django. For instance, there are great `django-authority` or `django-permissions` available out there.

---

<sup>1</sup> See <http://docs.djangoproject.com/en/1.2/topics/auth/#handling-object-permissions> for more detail.





---

## Installation

---

This application requires [Django 1.7](#) or higher and it is the only prerequisite before `django-guardian` may be used.

In order to install `django-guardian` simply use `pip`:

```
pip install django-guardian
```

or `easy_install`:

```
easy_install django-guardian
```

This would be enough to run `django-guardian`. However, in order to run tests or bounded example application, there are some other requirements. See more details about the topics:

- *Testing*
- *Example project*



---

## Configuration

---

After *installation* we can prepare our project for object permissions handling. In a settings module we need to add guardian to `INSTALLED_APPS`:

```
INSTALLED_APPS = (  
    # ...  
    'guardian',  
)
```

and hook guardian's authentication backend:

```
AUTHENTICATION_BACKENDS = (  
    'django.contrib.auth.backends.ModelBackend', # this is default  
    'guardian.backends.ObjectPermissionBackend',  
)
```

---

**Note:** Once project is configured to work with `django-guardian`, calling `syncdb` management command would create `User` instance for anonymous user support (with name of `AnonymousUser`).

---

**Note:** The Guardian anonymous user is different to the Django Anonymous user. The Django Anonymous user does not have an entry in the database, however the Guardian anonymous user does. This means that the following code will return an unexpected result:

```
from guardian.compat import get_user_model  
User = get_user_model()  
anon = User.get_anonymous()  
anon.is_anonymous() # returns False
```

---

We can change `id` to whatever we like. Project should be now ready to use object permissions.



---

## Optional settings

---

Guardian has following, optional configuration variables:

### 4.1 GUARDIAN\_RAISE\_403

New in version 1.0.4.

If set to `True`, guardian would raise `django.core.exceptions.PermissionDenied` error instead of returning empty `django.http.HttpResponseForbidden`.

**Warning:** Remember that you cannot use both `GUARDIAN_RENDER_403` AND `GUARDIAN_RAISE_403` - if both are set to `True`, `django.core.exceptions.ImproperlyConfigured` would be raised.

### 4.2 GUARDIAN\_RENDER\_403

New in version 1.0.4.

If set to `True`, guardian would try to render 403 response rather than return contentless `django.http.HttpResponseForbidden`. Would use template pointed by `GUARDIAN_TEMPLATE_403` to do that. Default is `False`.

**Warning:** Remember that you cannot use both `GUARDIAN_RENDER_403` AND `GUARDIAN_RAISE_403` - if both are set to `True`, `django.core.exceptions.ImproperlyConfigured` would be raised.

### 4.3 GUARDIAN\_TEMPLATE\_403

New in version 1.0.4.

Tells parts of guardian what template to use for responses with status code 403 (i.e. *permission\_required*). Defaults to `403.html`.

### 4.4 ANONYMOUS\_USER\_NAME

New in version 1.4.2.

This is the username of the anonymous user. Used to create the anonymous user and subsequently fetch the anonymous user as required.

If `ANONYMOUS_USER_NAME` is set to `None`, anonymous user object permissions are disabled. You may need to choose this option if creating a `User` object to represent anonymous users would be problematic in your environment.

Defaults to `"AnonymousUser"`.

**See also:**

<https://docs.djangoproject.com/en/1.5/topics/auth/customizing/#substituting-a-custom-user-model>

## 4.5 GUARDIAN\_GET\_INIT\_ANONYMOUS\_USER

New in version 1.2.

Guardian supports object level permissions for anonymous users, however when in our project we use custom `User` model, default function might fail. This can lead to issues as `guardian` tries to create anonymous user after each `syncdb` call. Object that is going to be created is retrieved using function pointed by this setting. Once retrieved, `save` method would be called on that instance.

Defaults to `"guardian.management.get_init_anonymous_user"`.

**See also:**

*Anonymous user creation*

## 5.1 Example project

Example project should be bundled with archive and be available at `example_project`. Before you can run it, some requirements have to be met. Those are easily installed using following command at example project's directory:

```
$ cd example_project
$ pip install -r requirements.txt
```

`django-guardian` from a directory above the `example_project` is automatically added to Python path at runtime.

And last thing before we can run example project is to create sqlite database:

```
$ ./manage.py migrate
```

Finally we can run dev server:

```
$ ./manage.py runserver
```

You should also create a user who can login to the admin site:

```
$ ./manage.py createsuperuser
```

Project is really basic and shows almost nothing but eventually it should expose some `django-guardian` functionality.

## 5.2 Assign object permissions

Assigning object permissions should be very simple once permissions are created for models.

### 5.2.1 Prepare permissions

Let's assume we have following model:

```
class Task(models.Model):
    summary = models.CharField(max_length=32)
    content = models.TextField()
    reported_by = models.ForeignKey(User)
    created_at = models.DateTimeField(auto_now_add=True)
```

... and we want to be able to set custom permission *view\_task*. We let Django know to do so by adding `permissions` tuple to `Meta` class and our final model could look like:

```
class Task(models.Model):
    summary = models.CharField(max_length=32)
    content = models.TextField()
    reported_by = models.ForeignKey(User)
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        permissions = (
            ('view_task', 'View task'),
        )
```

After we call `syncdb` (with a `--all` switch if you are using `south`) management command our *view\_task* permission would be added to default set of permissions.

---

**Note:** By default, Django adds 3 permissions for each registered model:

- *add\_modelname*
- *change\_modelname*
- *delete\_modelname*

(where *modelname* is a simplified name of our model's class). See <https://docs.djangoproject.com/en/dev/topics/auth/default/#default-permissions> for more detail.

---

There is nothing new here since creation of permissions is handled by `django`. Now we can move to *assigning object permissions*.

## 5.2.2 Assign object permissions

We can assign permissions for any user/group and object pairs using same, convenient function: `guardian.shortcuts.assign_perm()`.

### For user

Continuing our example we now can allow Joe user to view some task:

```
>>> from django.contrib.auth.models import User
>>> boss = User.objects.create(username='Big Boss')
>>> joe = User.objects.create(username='joe')
>>> task = Task.objects.create(summary='Some job', content='', reported_by=boss)
>>> joe.has_perm('view_task', task)
False
```

Well, not so fast Joe, let us create an object permission finally:

```
>>> from guardian.shortcuts import assign_perm
>>> assign_perm('view_task', joe, task)
>>> joe.has_perm('view_task', task)
True
```



## For group

This case doesn't really differ from user permissions assignment. The only difference is we have to pass Group instance rather than User.

```
>>> from django.contrib.auth.models import Group
>>> group = Group.objects.create(name='employees')
>>> assign_perm('change_task', group, task)
>>> joe.has_perm('change_task', task)
False
>>> # Well, joe is not yet within an *employees* group
>>> joe.groups.add(group)
>>> joe.has_perm('change_task', task)
True
```

Another example:

```
>>> from django.contrib.auth.models import User, Group
>>> from guardian.shortcuts import assign_perm
# fictional companies
>>> companyA = Company.objects.create(name="Company A")
>>> companyB = Company.objects.create(name="Company B")
# create groups
>>> companyUserGroupA = Group.objects.create(name="Company User Group A")
>>> companyUserGroupB = Group.objects.create(name="Company User Group B")
# assign object specific permissions to groups
>>> assign_perm('change_company', companyUserGroupA, companyA)
>>> assign_perm('change_company', companyUserGroupB, companyB)
# create user and add it to one group for testing
>>> userA = User.objects.create(username="User A")
>>> userA.groups.add(companyUserGroupA)
>>> userA.has_perm('change_company', companyA)
True
>>> userA.has_perm('change_company', companyB)
False
>>> userB = User.objects.create(username="User B")
>>> userB.has_perm('change_company', companyA)
False
>>> userA.has_perm('change_company', companyB)
True
```

## 5.2.3 Assigning Permissions inside Signals

Note that the Anonymous User is created before the Permissions are created. This may result in Django signals, e.g. `post_save` being sent before the Permissions are created. You will need to take this into an account when processing the signal.

```
@receiver(post_save, sender=User)
def user_post_save(sender, **kwargs):
    """
    Create a Profile instance for all newly created User instances. We only
    run on user creation to avoid having to check for existence on each call
    to User.save.
    """
    user, created = kwargs["instance"], kwargs["created"]
    if created and user.username != settings.ANONYMOUS_USER_NAME:
        from profiles.models import Profile
```

```
profile = Profile.objects.create(pk=user.pk, user=user, creator=user)
assign_perm("change_user", user, user)
assign_perm("change_profile", user, profile)
```

The check for `user.username != settings.ANONYMOUS_USER_NAME` is required otherwise the `assign_perm` calls will occur when the Anonymous User is created, however before there are any permissions available.

## 5.3 Check object permissions

Once we have *assigned some permissions*, we can get into detail about verifying permissions of a user or group.

### 5.3.1 Standard way

Normally to check if Joe is permitted to change Site objects we call `has_perm` method on an `User` instance:

```
>>> joe.has_perm('sites.change_site')
False
```

And for a specific `Site` instance we do the same but we pass `site` as additional argument:

```
>>> site = Site.objects.get_current()
>>> joe.has_perm('sites.change_site', site)
False
```

Let's assign permission and check again:

```
>>> from guardian.shortcuts import assign_perm
>>> assign_perm('sites.change_site', joe, site)
<UserObjectPermission: example.com | joe | change_site>
>>> joe = User.objects.get(username='joe')
>>> joe.has_perm('sites.change_site', site)
True
```

This uses the backend we have specified at settings module (see [Configuration](#)). More on the backend can be found at [Backend's API](#).

### 5.3.2 Inside views

Aside from the standard `has_perm` method, `django-guardian` provides some useful helpers for object permission checks.

#### get\_perms

To check permissions we can use a quick-and-dirty shortcut:

```
>>> from guardian.shortcuts import get_perms
>>>
>>> joe = User.objects.get(username='joe')
>>> site = Site.objects.get_current()
>>>
>>> 'change_site' in get_perms(joe, site)
True
```

It is probably better to use standard `has_perm` method. But for `Group` instances it is not as easy and `get_perms` could be handy here as it accepts both `User` and `Group` instances. If we need to do some more work, we can use lower level `ObjectPermissionChecker` class which is described in the next section.

There is also `get_user_perms` to get permissions assigned directly to the user (and not inherited from its superuser status or group membership). Similarly, `get_group_perms` returns only permissions which are inferred through user's group membership. `get_user_perms` and `get_group_perms` are useful when you care what permissions user has assigned, while `has_perm` is useful when you care about user's effective permissions.

## get\_objects\_for\_user

Sometimes there is a need to extract list of objects based on particular user, type of the object and provided permissions. For instance, lets say there is a `Project` model at `projects` application with custom `view_project` permission. We want to show our users projects they can actually *view*. This could be easily achieved using `get_objects_for_user`:

```
from django.shortcuts import render_to_response
from django.template import RequestContext
from projects.models import Project
from guardian.shortcuts import get_objects_for_user

def user_dashboard(request, template_name='projects/dashboard.html'):
    projects = get_objects_for_user(request.user, 'projects.view_project')
    return render_to_response(template_name, {'projects': projects},
                              RequestContext(request))
```

It is also possible to provide list of permissions rather than single string, own queryset (as `klass` argument) or control if result should be computed with (default) or without user's groups permissions.

### See also:

Documentation for `get_objects_for_user`

## ObjectPermissionChecker

At the core module of `django-guardian`, there is a `guardian.core.ObjectPermissionChecker` which checks permission of user/group for specific object. It caches results so it may be used at part of codes where we check permissions more than once.

Let's see it in action:

```
>>> joe = User.objects.get(username='joe')
>>> site = Site.objects.get_current()
>>> from guardian.core import ObjectPermissionChecker
>>> checker = ObjectPermissionChecker(joe) # we can pass user or group
>>> checker.has_perm('change_site', site)
True
>>> checker.has_perm('add_site', site) # no additional query made
False
>>> checker.get_perms(site)
[u'change_site']
```

## Using decorators

Standard `permission_required` decorator doesn't allow to check for object permissions. `django-guardian` is shipped with two decorators which may be helpful for simple object permission checks but remember that those

decorators hits database before decorated view is called - this means that if there is similar lookup made within a view then most probably one (or more, depending on lookups) extra database query would occur.

Let's assume we pass 'group\_name' argument to our view function which returns form to edit the group. Moreover, we want to return 403 code if check fails. This can be simply achieved using `permission_required_or_403` decorator:

```
>>> joe = User.objects.get(username='joe')
>>> foobars = Group.objects.create(name='foobars')
>>>
>>> from guardian.decorators import permission_required_or_403
>>> from django.http import HttpResponse
>>>
>>> @permission_required_or_403('auth.change_group',
>>>                             (Group, 'name', 'group_name'))
>>> def edit_group(request, group_name):
>>>     return HttpResponse('some form')
>>>
>>> from django.http import HttpRequest
>>> request = HttpRequest()
>>> request.user = joe
>>> edit_group(request, group_name='foobars')
<django.http.HttpResponseForbidden object at 0x102b43dd0>
>>>
>>> joe.groups.add(foobars)
>>> edit_group(request, group_name='foobars')
<django.http.HttpResponseForbidden object at 0x102b43e50>
>>>
>>> from guardian.shortcuts import assign_perm
>>> assign_perm('auth.change_group', joe, foobars)
<UserObjectPermission: foobars | joe | change_group>
>>>
>>> edit_group(request, group_name='foobars')
<django.http.HttpResponse object at 0x102b8c8d0>
>>> # Note that we now get normal HttpResponse, not forbidden
```

More on decorators can be read at corresponding [API page](#).

---

**Note:** Overall idea of decorators' lookups was taken from [django-authority](#) and all credits go to it's creator, Jannis Leidel.

---

### 5.3.3 Inside templates

django-guardian comes with special template tag `guardian.templatetags.guardian_tags.get_obj_perms()` which can store object permissions for a given user/group and instance pair. In order to use it we need to put following inside a template:

```
{% load guardian_tags %}
```

#### get\_obj\_perms

`guardian.templatetags.guardian_tags.get_obj_perms(parser, token)`

Returns a list of permissions (as codename strings) for a given user/group and obj (Model instance).

Parses `get_obj_perms` tag which should be in format:

```
{% get_obj_perms user/group for obj as "context_var" %}
```

**Note:** Make sure that you set and use those permissions in same template block (`{% block %}`).

Example of usage (assuming `flatpage` and `perm` objects are available from `context`):

```
{% get_obj_perms request.user for flatpage as "flatpage_perms" %}

{% if "delete_flatpage" in flatpage_perms %}
  <a href="/pages/delete?target={{ flatpage.url }}">Remove page</a>
{% endif %}
```

**Note:** Please remember that superusers would always get full list of permissions for a given object.

New in version 1.2.

As of v1.2, passing `None` as `obj` for this template tag won't raise obfuscated exception and would return empty permissions set instead.

## 5.4 Remove object permissions

Removing object permissions is as easy as assigning them. Just instead of `guardian.shortcuts.assign()` we would use `guardian.shortcuts.remove_perm()` function (it accepts same arguments).

### 5.4.1 Example

Let's get back to our fellow Joe:

```
>>> site = Site.object.get_current()
>>> joe.has_perm('change_site', site)
True
```

Now, simply remove this permission:

```
>>> from guardian.shortcuts import remove_perm
>>> remove_perm('change_site', joe, site)
>>> joe = User.objects.get(username='joe')
>>> joe.has_perm('change_site', site)
False
```

## 5.5 Admin integration

Django comes with excellent and widely used *Admin* application. Basically, it provides content management for Django applications. User with access to admin panel can manage users, groups, permissions and other data provided by system.

`django-guardian` comes with simple object permissions management integration for Django's admin application.

## 5.5.1 Usage

It is very easy to use admin integration. Simply use `GuardedModelAdmin` instead of standard `django.contrib.admin.ModelAdmin` class for registering models within the admin. In example, look at following model:

```
from django.db import models

class Post(models.Model):
    title = models.CharField('title', max_length=64)
    slug = models.SlugField(max_length=64)
    content = models.TextField('content')
    created_at = models.DateTimeField(auto_now_add=True, db_index=True)

    class Meta:
        permissions = (
            ('view_post', 'Can view post'),
        )
        get_latest_by = 'created_at'

    def __unicode__(self):
        return self.title

    @models.permalink
    def get_absolute_url(self):
        return {'post_slug': self.slug}
```

We want to register `Post` model within admin application. Normally, we would do this as follows within `admin.py` file of our application:

```
from django.contrib import admin

from posts.models import Post

class PostAdmin(admin.ModelAdmin):
    prepopulated_fields = {"slug": ("title",)}
    list_display = ('title', 'slug', 'created_at')
    search_fields = ('title', 'content')
    ordering = ('-created_at',)
    date_hierarchy = 'created_at'

admin.site.register(Post, PostAdmin)
```

If we would like to add object permissions management for `Post` model we would need to change `PostAdmin` base class into `GuardedModelAdmin`. Our code could look as follows:

```
from django.contrib import admin

from posts.models import Post

from guardian.admin import GuardedModelAdmin

class PostAdmin(GuardedModelAdmin):
    prepopulated_fields = {"slug": ("title",)}
    list_display = ('title', 'slug', 'created_at')
    search_fields = ('title', 'content')
```

```

ordering = ('-created_at',)
date_hierarchy = 'created_at'

admin.site.register(Post, PostAdmin)

```

And that's it. We can now navigate to **change** post page and just next to the *history* link we can click *Object permissions* button to manage row level permissions.

---

**Note:** Example above is shipped with `django-guardian` package with the example project.

---

## 5.6 Custom User model

New in version 1.1.

Django 1.5 comes with the ability to customize default `auth.User` model - either by subclassing `AbstractUser` or defining very own class. This can be very powerful, it must be done with caution, though. Basically, if we subclass `AbstractUser` or define many-to-many relation with `auth.Group` (and give reverse relate name **groups**) we should be fine.

By default `django-guardian` monkey patches the user model to add some needed functionality. This can result in errors if `guardian` is imported into the `models.py` of the same app where the custom user model lives.

To fix this, it is recommended to add the setting `GUARDIAN_MONKEY_PATCH = False` in your `settings.py` and add the `GuardianUserMixin` to your custom user model.

---

**Important:** `django-guardian` relies **heavily** on the `auth.User` model. Specifically it was build from the ground-up with relation between `auth.User` and `auth.Group` models. Retaining this relation is crucial for `guardian` - **without many to many User (custom or default) and auth.Group relation django-guardian will BREAK.**

---

**See also:**

Read more about customizing User model introduced in Django 1.5 here: <https://docs.djangoproject.com/en/1.5/topics/auth/customizing/#substituting-a-custom-user-model>.

### 5.6.1 Anonymous user creation

It is also possible to override default behavior of how instance for anonymous user is created. In example, let's imagine we have our user model as follows:

```

from django.contrib.auth.models import AbstractUser
from django.db import models

class CustomUser(AbstractUser):
    real_username = models.CharField(max_length=120, unique=True)
    birth_date = models.DateField() # field without default value

    USERNAME_FIELD = 'real_username'

```

Note that there is a `birth_date` field defined at the model and it does not have a default value. It would fail to create anonymous user instance as default implementation cannot know anything about `CustomUser` model.

In order to override the way anonymous instance is created we need to make `GUARDIAN_GET_INIT_ANONYMOUS_USER` pointing at our custom implementation. In example, let's define our init function:

```
import datetime

def get_anonymous_user_instance(User):
    return User(real_username='Anonymous', birth_date=datetime.date(1970, 1, 1))
```

and put it at `myapp/models.py`. Last step is to set proper configuration in our settings module:

```
GUARDIAN_GET_INIT_ANONYMOUS_USER = 'myapp.models.get_anonymous_user_instance'
```

## 5.7 Performance tuning

It is important to remember that by default `django-guardian` uses generic foreign keys to retain relation with any Django model. For most cases, it's probably good enough, however if we have a lot of queries being spanned and our database seems to be choking it might be a good choice to use *direct* foreign keys. Let's start with quick overview of how generic solution work and then we will move on to the tuning part.

### 5.7.1 Default, generic solution

`django-guardian` comes with two models: `UserObjectPermission` and `GroupObjectPermission`. They both have same, generic way of pointing to other models:

- `content_type` field telling what table (model class) target permission references to (ContentType instance)
- `object_pk` field storing value of target model instance primary key
- `content_object` field being a `GenericForeignKey`. Actually, it is not a foreign key in standard, relational database meaning - it is simply a proxy that can retrieve proper model instance being targeted by two previous fields

**See also:**

<https://docs.djangoproject.com/en/1.4/ref/contrib/contenttypes/#generic-relations>

Let's consider following model:

```
class Project(models.Model):
    name = models.CharField(max_length=128, unique=True)
```

In order to add a `change_project` permission for `joe` user we would use `assign_perm` shortcut:

```
>>> from guardian.shortcuts import assign_perm
>>> project = Project.objects.get(name='Foobar')
>>> joe = User.objects.get(username='joe')
>>> assign_perm('change_project', joe, project)
```

What it really does is: create an instance of `UserObjectPermission`. Something similar to:

```
>>> content_type = ContentType.objects.get_for_model(Project)
>>> perm = Permission.objects.get(content_type__app_label='app',
...     codename='change_project')
>>> UserObjectPermission.objects.create(user=joe, content_type=content_type,
...     permission=perm, object_pk=project.pk)
```



As there are no real foreign keys pointing at the target model, this solution might not be enough for all cases. For example, if we try to build an issues tracking service and we'd like to be able to support thousands of users and their project/tickets, object level permission checks can be slow with this generic solution.

## 5.7.2 Direct foreign keys

New in version 1.1.

In order to make our permission checks faster we can use direct foreign key solution. It actually is very simple to setup - we need to declare two new models next to our `Project` model, one for `User` and one for `Group` models:

```
from guardian.models import UserObjectPermissionBase
from guardian.models import GroupObjectPermissionBase

class Project(models.Model):
    name = models.CharField(max_length=128, unique=True)

class ProjectUserObjectPermission(UserObjectPermissionBase):
    content_object = models.ForeignKey(Project)

class ProjectGroupObjectPermission(GroupObjectPermissionBase):
    content_object = models.ForeignKey(Project)
```

---

**Important:** Name of the `ForeignKey` field is important and it should be `content_object` as underlying queries depends on it.

---

From now on, guardian will figure out that `Project` model has direct relation for user/group object permissions and will use those models. It is also possible to use only user or only group-based direct relation, however it is discouraged (it's not consistent and might be a quick road to hell from the maintainence point of view, especially).

---

**Note:** By defining direct relation models we can also tweak that object permission model, i.e. by adding some fields.

---

## 5.7.3 Prefetching permissions

New in version 1.4.3.

Naively looping through objects and checking permissions on each one using `has_perms` results in a permissions lookup in the database for each object. Large numbers of objects therefore produce large numbers of database queries which can considerably slow down your app. To avoid this, create an `ObjectPermissionChecker` and use its `prefetch_perms` method before looping through the objects. This will do a single lookup for all the objects and cache the results.

```
from guardian.core import ObjectPermissionChecker

joe = User.objects.get(username='joe')
projects = Project.objects.all()
checker = ObjectPermissionChecker(joe)

# Prefetch the permissions
checker.prefetch_perms(projects)

for project in projects:
```

```
# No additional lookups needed to check permissions
checker.has_perm('change_project', project)
```

## 5.8 Caveats

### 5.8.1 Orphaned object permissions

Note the following does not apply if using direct foreign keys, as documented in *Direct foreign keys*.

Permissions, including so called *per object permissions*, are sometimes tricky to manage. One case is how we can manage permissions that are no longer used. Normally, there should be no problems, however with some particular setup it is possible to reuse primary keys of database models which were used in the past once. We will not answer how bad such situation can be - instead we will try to cover how we can deal with this.

Let's imagine our table has primary key to the filesystem path. We have a record with pk equal to `/home/www/joe.config`. User *jane* has read access to joe's configuration and we store that information in database by creating guardian's object permissions. Now, *joe* user removes account from our site and another user creates account with *joe* as username. The problem is that if we haven't removed object permissions explicitly in the process of first *joe* account removal, *jane* still has read permissions for *joe*'s configuration file - but this is another user.

There is no easy way to deal with orphaned permissions as they are not foreign keyed with objects directly. Even if they would, there are some database engines - or *ON DELETE* rules - which restricts removal of related objects.

---

**Important:** It is **extremely** important to remove *UserObjectPermission* and *GroupObjectPermission* as we delete objects for which permissions are defined.

---

Guardian comes with utility function which tries to help to remove orphaned object permissions. Remember - those are only helpers. Applications should remove those object permissions explicitly by itself.

Taking our previous example, our application should remove user object for *joe*, however, permissions for *joe* user assigned to *jane* would **NOT** be removed. In this case, it would be very easy to remove user/group object permissions if we connect proper action with proper signal. This could be achieved by following snippet:

```
from django.contrib.contenttypes.models import ContentType
from django.db.models import Q
from django.db.models.signals import pre_delete
from guardian.models import User
from guardian.models import UserObjectPermission
from guardian.models import GroupObjectPermission

def remove_obj_perms_connected_with_user(sender, instance, **kwargs):
    filters = Q(content_type=ContentType.objects.get_for_model(instance),
                object_pk=instance.pk)
    UserObjectPermission.objects.filter(filters).delete()
    GroupObjectPermission.objects.filter(filters).delete()

pre_delete.connect(remove_obj_perms_connected_with_user, sender=User)
```

This signal handler would remove all object permissions connected with user just before user is actually removed.

If we forgot to add such handlers, we may still remove orphaned object permissions by using *clean\_orphan\_obj\_perms* command. If our application uses *celery*, it is also very easy to remove orphaned permissions periodically with *guardian.utils.clean\_orphan\_obj\_perms()* function. We would still

**strongly** advise to remove orphaned object permissions explicitly (i.e. at view that confirms object removal or using signals as described above).

**See also:**

- `guardian.utils.clean_orphan_obj_perms()`
- `clean_orphan_obj_perms`

## 5.8.2 Using multiple databases

This is not supported at present time due to a Django bug. See [288](#) and [16281](#).



## 6.1 Admin

### 6.1.1 GuardedModelAdmin

**class** guardian.admin.**GuardedModelAdmin** (*model, admin\_site*)

Extends `django.contrib.admin.ModelAdmin` class. Provides some extra views for object permissions management at admin panel. It also changes default `change_form_template` option to `'admin/guardian/model/change_form.html'` which is required for proper url (object permissions related) being shown at the model pages.

#### Extra options

`GuardedModelAdmin.obj_perms_manage_template`

*Default:* `admin/guardian/model/obj_perms_manage.html`

`GuardedModelAdmin.obj_perms_manage_user_template`

*Default:* `admin/guardian/model/obj_perms_manage_user.html`

`GuardedModelAdmin.obj_perms_manage_group_template`

*Default:* `admin/guardian/model/obj_perms_manage_group.html`

`GuardedModelAdmin.user_can_access_owned_objects_only`

*Default:* `False`

If this would be set to `True`, `request.user` would be used to filter out objects he or she doesn't own (checking `user` field of used model - field name may be overridden by `user_owned_objects_field` option).

---

**Note:** Please remember that this will **NOT** affect superusers! Admins would still see all items.

---

`GuardedModelAdmin.user_can_access_owned_by_group_objects_only`

*Default:* `False`

If this would be set to `True`, `request.user` would be used to filter out objects her or his group doesn't own (checking if any group user belongs to is set as `group` field of the object; name of the field can be changed by overriding `group_owned_objects_field`).

---

**Note:** Please remember that this will **NOT** affect superusers! Admins would still see all items.

---

`GuardedModelAdmin.group_owned_objects_field`

*Default:* `group`

`GuardedModelAdmin.include_object_permissions_urls`

*Default:* `True`

New in version 1.2.

Might be set to `False` in order **NOT** to include guardian-specific urls.

### Usage example

Just use `GuardedModelAdmin` instead of `django.contrib.admin.ModelAdmin`.

```
from django.contrib import admin
from guardian.admin import GuardedModelAdmin
from myapp.models import Author

class AuthorAdmin(GuardedModelAdmin):
    pass

admin.site.register(Author, AuthorAdmin)
```

## 6.2 Backends

### 6.2.1 ObjectPermissionBackend

`class guardian.backends.ObjectPermissionBackend`

**get\_all\_permissions** (*user\_obj*, *obj=None*)

Returns a set of permission strings that the given *user\_obj* has for *obj*

**has\_perm** (*user\_obj*, *perm*, *obj=None*)

Returns `True` if given *user\_obj* has *perm* for *obj*. If no *obj* is given, `False` is returned.

---

**Note:** Remember, that if user is not *active*, all checks would return `False`.

---

Main difference between Django's `ModelBackend` is that we can pass *obj* instance here and *perm* doesn't have to contain *app\_label* as it can be retrieved from given *obj*.

#### Inactive user support

If user is authenticated but inactive at the same time, all checks always returns `False`.

## 6.3 Core

### 6.3.1 ObjectPermissionChecker

`class guardian.core.ObjectPermissionChecker (user_or_group=None)`  
 Generic object permissions checker class being the heart of django-guardian.

---

**Note:** Once checked for single object, permissions are stored and we don't hit database again if another check is called for this object. This is great for templates, views or other request based checks (assuming we don't have hundreds of permissions on a single object as we fetch all permissions for checked object).

On the other hand, if we call `has_perm` for `perm1/object1`, then we change permission state and call `has_perm` again for same `perm1/object1` on same instance of `ObjectPermissionChecker` we won't see a difference as permissions are already fetched and stored within cache dictionary.

---

Constructor for `ObjectPermissionChecker`.

**Parameters** `user_or_group` – should be an `User`, `AnonymousUser` or `Group` instance

`get_local_cache_key (obj)`  
 Returns cache key for `_obj_perms_cache` dict.

`get_perms (obj)`  
 Returns list of `codename`'s of all permissions for given `obj`.

**Parameters** `obj` – Django model instance for which permission should be checked

`has_perm (perm, obj)`  
 Checks if user/group has given permission for object.

**Parameters**

- `perm` – permission as string, may or may not contain `app_label` prefix (if not prefixed, we grab `app_label` from `obj`)
- `obj` – Django model instance for which permission should be checked

`prefetch_perms (objects)`  
 Prefetches the permissions for objects in `objects` and puts them in the cache.

**Parameters** `objects` – Iterable of Django model objects

## 6.4 Decorators

### 6.4.1 permission\_required

`guardian.decorators.permission_required (perm, lookup_variables=None, **kwargs)`  
 Decorator for views that checks whether a user has a particular permission enabled.

Optionally, instances for which check should be made may be passed as a second argument or as a tuple parameters same as those passed to `get_object_or_404` but must be provided as pairs of strings. This way decorator can fetch i.e. `User` instance based on performed request and check permissions on it (without this, one would need to fetch user instance at view's logic and check permission inside a view).

**Parameters**

- **login\_url** – if denied, user would be redirected to location set by this parameter. Defaults to `django.conf.settings.LOGIN_URL`.
- **redirect\_field\_name** – name of the parameter passed if redirected. Defaults to `django.contrib.auth.REDIRECT_FIELD_NAME`.
- **return\_403** – if set to `True` then instead of redirecting to the login page, response with status code 403 is returned ( `django.http.HttpResponseForbidden` instance or rendered template - see `GUARDIAN_RENDER_403`). Defaults to `False`.
- **accept\_global\_perms** – if set to `True`, then *object level permission* would be required **only if user does NOT have global permission** for target *model*. If turned on, makes this decorator like an extension over standard `django.contrib.admin.decorators.permission_required` as it would check for global permissions first. Defaults to `False`.

Examples:

```
@permission_required('auth.change_user', return_403=True)
def my_view(request):
    return HttpResponse('Hello')

@permission_required('auth.change_user', (User, 'username', 'username'))
def my_view(request, username):
    """
    auth.change_user permission would be checked based on given
    'username'. If view's parameter would be named ``name``, we would
    rather use following decorator::

        @permission_required('auth.change_user', (User, 'username', 'name'))
    """
    user = get_object_or_404(User, username=username)
    return user.get_absolute_url()

@permission_required('auth.change_user',
                    (User, 'username', 'username', 'groups__name', 'group_name'))
def my_view(request, username, group_name):
    """
    Similar to the above example, here however we also make sure that
    one of user's group is named same as request's ``group_name`` param.
    """
    user = get_object_or_404(User, username=username,
                            group__name=group_name)
    return user.get_absolute_url()
```

## 6.4.2 permission\_required\_or\_403

`guardian.decorators.permission_required_or_403` (*perm, \*args, \*\*kwargs*)

Simple wrapper for `permission_required` decorator.

Standard Django's `permission_required` decorator redirects user to login page in case permission check failed. This decorator may be used to return `HttpResponseForbidden` (status 403) instead of redirection.

The only difference between `permission_required` decorator is that this one always set `return_403` parameter to `True`.



## 6.5 Forms

### 6.5.1 UserObjectPermissionsForm

**class** guardian.forms.**UserObjectPermissionsForm**(user, \*args, \*\*kwargs)

Bases: guardian.forms.BaseObjectPermissionsForm

Object level permissions management form for usage with User instances.

Example usage:

```
from django.shortcuts import get_object_or_404
from myapp.models import Post
from guardian.forms import UserObjectPermissionsForm
from django.contrib.auth.models import User

def my_view(request, post_slug, user_id):
    user = get_object_or_404(User, id=user_id)
    post = get_object_or_404(Post, slug=post_slug)
    form = UserObjectPermissionsForm(user, post, request.POST or None)
    if request.method == 'POST' and form.is_valid():
        form.save_obj_perms()
    ...
```

**save\_obj\_perms** ()

Saves selected object permissions by creating new ones and removing those which were not selected but already exists.

Should be called *after* form is validated.

### 6.5.2 GroupObjectPermissionsForm

**class** guardian.forms.**GroupObjectPermissionsForm**(group, \*args, \*\*kwargs)

Bases: guardian.forms.BaseObjectPermissionsForm

Object level permissions management form for usage with Group instances.

Example usage:

```
from django.shortcuts import get_object_or_404
from myapp.models import Post
from guardian.forms import GroupObjectPermissionsForm
from guardian.models import Group

def my_view(request, post_slug, group_id):
    group = get_object_or_404(Group, id=group_id)
    post = get_object_or_404(Post, slug=post_slug)
    form = GroupObjectPermissionsForm(group, post, request.POST or None)
    if request.method == 'POST' and form.is_valid():
        form.save_obj_perms()
    ...
```

**save\_obj\_perms** ()

Saves selected object permissions by creating new ones and removing those which were not selected but already exists.

Should be called *after* form is validated.

### 6.5.3 BaseObjectPermissionsForm

**class** `guardian.forms.BaseObjectPermissionsForm` (*obj*, \**args*, \*\**kwargs*)  
Base form for object permissions management. Needs to be extended for usage with users and/or groups.

Constructor for BaseObjectPermissionsForm.

**Parameters** *obj* – Any instance which form would use to manage object permissions”

**are\_obj\_perms\_required()**  
Indicates if at least one object permission should be required. Default: `False`.

**get\_obj\_perms\_field()**  
Returns field instance for object permissions management. May be replaced entirely.

**get\_obj\_perms\_field\_choices()**  
Returns choices for object permissions management field. Default: list of tuples (*codename*, *name*) for each `Permission` instance for the managed object.

**get\_obj\_perms\_field\_class()**  
Returns object permissions management field’s base class. Default: `django.forms.MultipleChoiceField`.

**get\_obj\_perms\_field\_initial()**  
Returns initial object permissions management field choices. Default: `[]` (empty list).

**get\_obj\_perms\_field\_label()**  
Returns label of the object permissions management field. Default: `_("Permissions")` (marked to be translated).

**get\_obj\_perms\_field\_name()**  
Returns name of the object permissions management field. Default: `permission`.

**get\_obj\_perms\_field\_widget()**  
Returns object permissions management field’s widget base class. Default: `django.forms.SelectMultiple`.

**save\_obj\_perms()**  
Must be implemented in concrete form class. This method should store selected object permissions.

## 6.6 Management commands

**class** `guardian.management.commands.clean_orphan_obj_perms.Command` (*stdout=None*,  
*stderr=None*,  
*no\_color=False*)  
`clean_orphan_obj_perms` command is a tiny wrapper around `guardian.utils.clean_orphan_obj_perms()`.

Usage:

```
$ python manage.py clean_orphan_obj_perms
Removed 11 object permission entries with no targets
```

## 6.7 Managers

### 6.7.1 UserObjectPermissionManager

**class** `guardian.managers.UserObjectPermissionManager`

## 6.7.2 GroupObjectPermissionManager

```
class guardian.managers.GroupObjectPermissionManager
```

## 6.8 Mixins

New in version 1.0.4.

### 6.8.1 LoginRequiredMixin

```
class guardian.mixins.LoginRequiredMixin
```

A login required mixin for use with class based views. This Class is a light wrapper around the *login\_required* decorator and hence function parameters are just attributes defined on the class.

Due to parent class order traversal this mixin must be added as the left most mixin of a view.

The mixin has exactly the same flow as *login\_required* decorator:

If the user isn't logged in, redirect to `settings.LOGIN_URL`, passing the current absolute path in the query string. Example: `/accounts/login/?next=/polls/3/`.

If the user is logged in, execute the view normally. The view code is free to assume the user is logged in.

#### Class Settings

```
LoginRequiredMixin.redirect_field_name
```

*Default:* 'next'

```
LoginRequiredMixin.login_url
```

*Default:* settings.LOGIN\_URL

### 6.8.2 PermissionRequiredMixin

```
class guardian.mixins.PermissionRequiredMixin
```

A view mixin that verifies if the current logged in user has the specified permission by wrapping the `request.user.has_perm(..)` method.

If a *get\_object()* method is defined either manually or by including another mixin (for example *SingleObjectMixin*) or `self.object` is defined then the permission will be tested against that specific instance, alternatively you can specify *get\_permission\_object()* method if `self.object` or *get\_object()* does not return the object against you want to test permission

The mixin does the following:

If the user isn't logged in, redirect to `settings.LOGIN_URL`, passing the current absolute path in the query string. Example: `/accounts/login/?next=/polls/3/`.

If the *raise\_exception* is set to True than rather than redirect to login page a *PermissionDenied* (403) is raised.

If the user is logged in, and passes the permission check than the view is executed normally.

#### Example Usage:

```
class SecureView(PermissionRequiredMixin, View):
    ...
    permission_required = 'auth.change_user'
    ...
```

### Class Settings

`PermissionRequiredMixin.permission_required`

*Default:* None, must be set to either a string or list of strings in format: `<app_label>.<permission_codename>`.

`PermissionRequiredMixin.login_url`

*Default:* `settings.LOGIN_URL`

`PermissionRequiredMixin.redirect_field_name`

*Default:* `'next'`

`PermissionRequiredMixin.return_403`

*Default:* False. Returns 403 error page instead of redirecting user.

`PermissionRequiredMixin.raise_exception`

*Default:* False

***permission\_required*** - the permission to check of form “`<app_label>.<permission codename>`” i.e. `'polls.can_vote'` for a permission on a model in the polls application.

`PermissionRequiredMixin.accept_global_perms`

***Default:*** False, If `accept_global_perms` would be set to True, then mixing would first check for global perms, if none found, then it will proceed to check object level permissions.

**`PermissionRequiredMixin.permission_object`** *Default:* (not set), object against which test the permission; if not set fallback to `self.get_permission_object()` which return `self.get_object()` or `self.object` by default.

**`check_permissions`** (*request*)

Checks if *request.user* has all permissions returned by `get_required_permissions` method.

**Parameters** *request* – Original request.

**`get_required_permissions`** (*request=None*)

Returns list of permissions in format `<app_label>.<codename>` that should be checked against *request.user* and *object*. By default, it returns list from `permission_required` attribute.

**Parameters** *request* – Original request.

**`on_permission_check_fail`** (*request, response, obj=None*)

Method called upon permission check fail. By default it does nothing and should be overridden, if needed.

**Parameters**

- **request** – Original request
- **response** – 403 response returned by `check_permissions` method.
- **obj** – Object that was fetched from the view (using `get_object` method or `object` attribute, in that order).

## 6.9 Models

### 6.9.1 BaseObjectPermission

`class guardian.models.BaseObjectPermission(*args, **kwargs)`  
 Abstract ObjectPermission class. Actual class should additionally define a `content_object` field and either `user` or `group` field.

### 6.9.2 UserObjectPermission

`class guardian.models.UserObjectPermission(id, permission, content_type, object_pk, user)`

### 6.9.3 GroupObjectPermission

`class guardian.models.GroupObjectPermission(id, permission, content_type, object_pk, group)`

## 6.10 Shortcuts

Convenient shortcuts to manage or check object permissions.

### 6.10.1 assign\_perm

`guardian.shortcuts.assign_perm(perm, user_or_group, obj=None)`  
 Assigns permission to user/group and object pair.

#### Parameters

- **perm** – proper permission for given obj, as string (in format: `app_label.codename` or `codename`) or `Permission` instance. If `obj` is not given, must be in format `app_label.codename` or `Permission` instance.
- **user\_or\_group** – instance of `User`, `AnonymousUser` or `Group`; passing any other object would raise `guardian.exceptions.NotUserNorGroup` exception
- **obj** – persisted Django's `Model` instance or `QuerySet` of Django `Model` instances or `None` if assigning global permission. Default is `None`.

We can assign permission for `Model` instance for specific user:

```
>>> from django.contrib.sites.models import Site
>>> from guardian.models import User
>>> from guardian.shortcuts import assign_perm
>>> site = Site.objects.get_current()
>>> user = User.objects.create(username='joe')
>>> assign_perm("change_site", user, site)
<UserObjectPermission: example.com | joe | change_site>
>>> user.has_perm("change_site", site)
True
```

... or we can assign permission for group:

```
>>> group = Group.objects.create(name='joe-group')
>>> user.groups.add(group)
>>> assign_perm("delete_site", group, site)
<GroupObjectPermission: example.com | joe-group | delete_site>
>>> user.has_perm("delete_site", site)
True
```

### Global permissions

This function may also be used to assign standard, *global* permissions if `obj` parameter is omitted. Added Permission would be returned in that case:

```
>>> assign_perm("sites.change_site", user)
<Permission: sites | site | Can change site>
```

## 6.10.2 remove\_perm

`guardian.shortcuts.remove_perm` (*perm*, *user\_or\_group=None*, *obj=None*)

Removes permission from user/group and object pair.

### Parameters

- **perm** – proper permission for given `obj`, as string (in format: `app_label.codename` or `codename`). If `obj` is not given, must be in format `app_label.codename`.
- **user\_or\_group** – instance of `User`, `AnonymousUser` or `Group`; passing any other object would raise `guardian.exceptions.NotUserNorGroup` exception
- **obj** – persisted Django's `Model` instance or `QuerySet` of Django `Model` instances or `None` if assigning global permission. Default is `None`.

## 6.10.3 get\_perms

`guardian.shortcuts.get_perms` (*user\_or\_group*, *obj*)

Returns permissions for given user/group and object pair, as list of strings.

## 6.10.4 get\_user\_perms

`guardian.shortcuts.get_user_perms` (*user*, *obj*)

Returns permissions for given user and object pair, as list of strings, only those assigned directly for the user.

## 6.10.5 get\_group\_perms

`guardian.shortcuts.get_group_perms` (*user\_or\_group*, *obj*)

Returns permissions for given user/group and object pair, as list of strings. It returns only those which are inferred through groups.

## 6.10.6 get\_perms\_for\_model

`guardian.shortcuts.get_perms_for_model` (*cls*)

Returns queryset of all `Permission` objects for the given class. It is possible to pass `Model` as class or instance.

## 6.10.7 get\_users\_with\_perms

`guardian.shortcuts.get_users_with_perms` (*obj*, *attach\_perms=False*, *with\_superuser=False*, *with\_group\_users=True*)

Returns queryset of all User objects with *any* object permissions for the given *obj*.

### Parameters

- **obj** – persisted Django’s Model instance
- **attach\_perms** – Default: False. If set to True result would be dictionary of User instances with permissions’ codenames list as values. This would fetch users eagerly!
- **with\_superuser** – Default: False. If set to True result would contain all superusers.
- **with\_group\_users** – Default: True. If set to False result would **not** contain those users who have only group permissions for given *obj*.

Example:

```
>>> from django.contrib.flatpages.models import FlatPage
>>> from django.contrib.auth.models import User
>>> from guardian.shortcuts import assign_perm, get_users_with_perms
>>>
>>> page = FlatPage.objects.create(title='Some page', path='/some/page/')
>>> joe = User.objects.create_user('joe', 'joe@example.com', 'joesecret')
>>> assign_perm('change_flatpage', joe, page)
>>>
>>> get_users_with_perms(page)
[<User: joe>]
>>>
>>> get_users_with_perms(page, attach_perms=True)
{<User: joe>: [u'change_flatpage']}
```

## 6.10.8 get\_groups\_with\_perms

`guardian.shortcuts.get_groups_with_perms` (*obj*, *attach\_perms=False*)

Returns queryset of all Group objects with *any* object permissions for the given *obj*.

### Parameters

- **obj** – persisted Django’s Model instance
- **attach\_perms** – Default: False. If set to True result would be dictionary of Group instances with permissions’ codenames list as values. This would fetch groups eagerly!

Example:

```
>>> from django.contrib.flatpages.models import FlatPage
>>> from guardian.shortcuts import assign_perm, get_groups_with_perms
>>> from guardian.models import Group
>>>
>>> page = FlatPage.objects.create(title='Some page', path='/some/page/')
>>> admins = Group.objects.create(name='Admins')
>>> assign_perm('change_flatpage', admins, page)
>>>
>>> get_groups_with_perms(page)
[<Group: admins>]
>>>
```

```
>>> get_groups_with_perms(page, attach_perms=True)
{<Group: admins>: [u'change_flatpage']}
```

### 6.10.9 get\_objects\_for\_user

`guardian.shortcuts.get_objects_for_user`(*user*, *perms*, *klass=None*, *use\_groups=True*, *any\_perm=False*, *with\_superuser=True*, *accept\_global\_perms=True*)

Returns queryset of objects for which a given user has *all* permissions present at *perms*.

If *perms* is an empty list, then it returns objects for which a given user has *any* object permission.

#### Parameters

- **user** – User or AnonymousUser instance for which objects would be returned.
- **perms** – single permission string, or sequence of permission strings which should be checked. If *klass* parameter is not given, those should be full permission names rather than only codenames (i.e. `auth.change_user`). If more than one permission is present within sequence, their content type **must** be the same or `MixedContentTypeError` exception would be raised.
- **klass** – may be a Model, Manager or QuerySet object. If not given this parameter would be computed based on given params.
- **use\_groups** – if `False`, wouldn't check user's groups object permissions. Default is `True`.
- **any\_perm** – if `True`, any of permission in sequence is accepted. Default is `False`.
- **with\_superuser** – if `True` and if `user.is_superuser` is set, returns the entire queryset. Otherwise will only return objects the user has explicit permissions. This must be `True` for the `accept_global_perms` parameter to have any affect. Default is `True`.
- **accept\_global\_perms** – if `True` takes global permissions into account. Object based permissions are taken into account if more than one permission is handed in in *perms* and at least one of these perms is not globally set. If `any_perm` is set to `false` then the intersection of matching object is returned. Note, that if `with_superuser` is `False`, `accept_global_perms` will be ignored, which means that only object permissions will be checked! Default is `True`.

#### Raises

- **MixedContentTypeError** – when computed content type for *perms* and/or *klass* clashes.
- **WrongAppError** – if cannot compute app label for given *perms*/*klass*.

Example:

```
>>> from django.contrib.auth.models import User
>>> from guardian.shortcuts import get_objects_for_user
>>> joe = User.objects.get(username='joe')
>>> get_objects_for_user(joe, 'auth.change_group')
[]
>>> from guardian.shortcuts import assign_perm
>>> group = Group.objects.create('some group')
>>> assign_perm('auth.change_group', joe, group)
>>> get_objects_for_user(joe, 'auth.change_group')
[<Group some group>]
```



The permission string can also be an iterable. Continuing with the previous example:

```
>>> get_objects_for_user(joe, ['auth.change_group', 'auth.delete_group'])
[]
>>> get_objects_for_user(joe, ['auth.change_group', 'auth.delete_group'], any_perm=True)
[<Group some group>]
>>> assign_perm('auth.delete_group', joe, group)
>>> get_objects_for_user(joe, ['auth.change_group', 'auth.delete_group'])
[<Group some group>]
```

Take global permissions into account:

```
>>> jack = User.objects.get(username='jack')
>>> assign_perm('auth.change_group', jack) # this will set a global permission
>>> get_objects_for_user(jack, 'auth.change_group')
[<Group some group>]
>>> group2 = Group.objects.create('other group')
>>> assign_perm('auth.delete_group', jack, group2)
>>> get_objects_for_user(jack, ['auth.change_group', 'auth.delete_group']) # this retrieves inter
[<Group other group>]
>>> get_objects_for_user(jack, ['auth.change_group', 'auth.delete_group'], any_perm=True) # this retr
[<Group some group>, <Group other group>]
```

If `accept_global_perms` is set to `True`, then all assigned global permissions will also be taken into account.

- Scenario 1: a user has view permissions generally defined on the model ‘books’ but no object based permission on a single book instance:
  - If `accept_global_perms` is `True`: List of all books will be returned.
  - If `accept_global_perms` is `False`: list will be empty.
- Scenario 2: a user has view permissions generally defined on the model ‘books’ and also has an object based permission to view book ‘Whatever’:
  - If `accept_global_perms` is `True`: List of all books will be returned.
  - If `accept_global_perms` is `False`: list will only contain book ‘Whatever’.
- Scenario 3: a user only has object based permission on book ‘Whatever’:
  - If `accept_global_perms` is `True`: List will only contain book ‘Whatever’.
  - If `accept_global_perms` is `False`: List will only contain book ‘Whatever’.
- Scenario 4: a user does not have any permission:
  - If `accept_global_perms` is `True`: Empty list.
  - If `accept_global_perms` is `False`: Empty list.

### 6.10.10 get\_objects\_for\_group

`guardian.shortcuts.get_objects_for_group(group, perms, klass=None, any_perm=False, accept_global_perms=True)`

Returns queryset of objects for which a given `group` has *all* permissions present at `perms`.

#### Parameters

- **group** – Group instance for which objects would be returned.
- **perms** – single permission string, or sequence of permission strings which should be checked. If `klass` parameter is not given, those should be full permission names rather

than only codenames (i.e. `auth.change_user`). If more than one permission is present within sequence, their content type **must** be the same or `MixedContentTypeError` exception would be raised.

- **class** – may be a Model, Manager or QuerySet object. If not given this parameter would be computed based on given params.
- **any\_perm** – if True, any of permission in sequence is accepted
- **accept\_global\_perms** – if True takes global permissions into account. If any\_perm is set to false then the intersection of matching objects based on global and object based permissions is returned. Default is True.

#### Raises

- **MixedContentTypeError** – when computed content type for perms and/or class clashes.
- **WrongAppError** – if cannot compute app label for given perms/ class.

Example:

Let's assume we have a Task model belonging to the `tasker` app with the default `add_task`, `change_task` and `delete_task` permissions provided by Django:

```
>>> from guardian.shortcuts import get_objects_for_group
>>> from tasker import Task
>>> group = Group.objects.create('some group')
>>> task = Task.objects.create('some task')
>>> get_objects_for_group(group, 'tasker.add_task')
[]
>>> from guardian.shortcuts import assign_perm
>>> assign_perm('tasker.add_task', group, task)
>>> get_objects_for_group(group, 'tasker.add_task')
[<Task some task>]
```

The permission string can also be an iterable. Continuing with the previous example:

```
>>> get_objects_for_group(group, ['tasker.add_task', 'tasker.delete_task'])
[]
>>> assign_perm('tasker.delete_task', group, task)
>>> get_objects_for_group(group, ['tasker.add_task', 'tasker.delete_task'])
[<Task some task>]
```

Global permissions assigned to the group are also taken into account. Continuing with previous example:

```
>>> task_other = Task.objects.create('other task')
>>> assign_perm('tasker.change_task', group)
>>> get_objects_for_group(group, ['tasker.change_task'])
[<Task some task>, <Task other task>]
>>> get_objects_for_group(group, ['tasker.change_task'], accept_global_perms=False)
[<Task some task>]
```

## 6.11 Utilities

django-guardian helper functions.

Functions defined within this module should be considered as django-guardian's internal functionality. They are **not** guaranteed to be stable - which means they actual input parameters/output type may change in future releases.

### 6.11.1 get\_anonymous\_user

`guardian.utils.get_anonymous_user()`

Returns `User` instance (not `AnonymousUser`) depending on `ANONYMOUS_USER_NAME` configuration.

### 6.11.2 get\_identity

`guardian.utils.get_identity(identity)`

Returns `(user_obj, None)` or `(None, group_obj)` tuple depending on what is given. Also accepts `AnonymousUser` instance but would return `User` instead - it is convenient and needed for authorization backend to support anonymous users.

**Parameters** `identity` – either `User` or `Group` instance

**Raises** `NotUserNorGroup` – if cannot return proper identity instance

**Examples:**

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create(username='joe')
>>> get_identity(user)
(<User: joe>, None)

>>> group = Group.objects.create(name='users')
>>> get_identity(group)
(None, <Group: users>)

>>> anon = AnonymousUser()
>>> get_identity(anon)
(<User: AnonymousUser>, None)

>>> get_identity("not instance")
...
NotUserNorGroup: User/AnonymousUser or Group instance is required (got )
```

### 6.11.3 clean\_orphan\_obj\_perms

`guardian.utils.clean_orphan_obj_perms()`

Seeks and removes all object permissions entries pointing at non-existing targets.

Returns number of removed objects.

## 6.12 Template tags

django-guardian template tags. To use in a template just put the following `load` tag inside a template:

```
{% load guardian_tags %}
```

## 6.12.1 get\_obj\_perms

guardian.templatetags.guardian\_tags.get\_obj\_perms (parser, token)

Returns a list of permissions (as codename strings) for a given user/group and obj (Model instance).

Parses get\_obj\_perms tag which should be in format:

```
{% get_obj_perms user/group for obj as "context_var" %}
```

---

**Note:** Make sure that you set and use those permissions in same template block (`{% block %}`).

---

Example of usage (assuming flatpage and perm objects are available from *context*):

```
{% get_obj_perms request.user for flatpage as "flatpage_perms" %}

{% if "delete_flatpage" in flatpage_perms %}
  <a href="/pages/delete?target={{ flatpage.url }}">Remove page</a>
{% endif %}
```

---

**Note:** Please remember that superusers would always get full list of permissions for a given object.

---

New in version 1.2.

As of v1.2, passing `None` as `obj` for this template tag won't rise obfuscated exception and would return empty permissions set instead.

## 7.1 Overview

Here we describe the development process overview. It's in F.A.Q. format to make it simple.

### 7.1.1 Why devel is default branch?

Since version 1.2 we try to make `master` in a production-ready state. It does NOT mean it is production ready, but it SHOULD be. In example, tests at `master` should always pass. Actually, whole tox suite should pass. And it's test coverage should be at 100% level.

`devel` branch, on the other hand, can break. It shouldn't but it is acceptable. As a user, you should NEVER use non-master branches at production. All the changes are pushed from `devel` to `master` before next release. It might happen more frequently.

### 7.1.2 How to file a ticket?

Just go to <https://github.com/django-guardian/django-guardian/issues> and create new one.

### 7.1.3 How do I get involved?

It's simple! If you want to fix a bug, extend documentation or whatever you think is appropriate for the project and involves changes, just fork the project at github (<https://github.com/django-guardian/django-guardian>), create a separate branch, hack on it, publish changes at your fork and create a pull request.

Here is a quick how to:

1. Fork a project: <https://github.com/django-guardian/django-guardian/fork>
2. Checkout project to your local machine:

```
$ git clone git@github.com:YOUR_NAME/django-guardian.git
```

3. Create a new branch with name describing change you are going to work on:

```
$ git checkout -b bugfix/support-for-custom-model
```

4. Perform changes at newly created branch. Remember to include tests (if this is code related change) and run test suite. See [running tests documentation](#). Also, remember to add yourself to the `AUTHORS` file.

- (Optional) Squash commits. If you have multiple commits and it doesn't make much sense to have them separated (and it usually makes little sense), please consider merging all changes into single commit. Please see <https://help.github.com/articles/interactive-rebase> if you need help with that.
- Publish changes:

```
$ git push origin YOUR_BRANCH_NAME
```

- Create a Pull Request (<https://help.github.com/articles/creating-a-pull-request>). Usually it's as simple as opening up [https://github.com/YOUR\\_NAME/django-guardian](https://github.com/YOUR_NAME/django-guardian) and clicking on review button for newly created branch. There you can make final review of your changes and if everything seems fine, create a Pull Request.

## 7.1.4 Why my issue/pull request was closed?

We usually put an explanation while we close issue or PR. It might be for various reasons, i.e. there were no reply for over a month after our last comment, there were no tests for the changes etc.

## 7.2 Testing

### 7.2.1 Introduction

django-guardian is extending capabilities of Django's authorization facilities and as so, it changes it's security somehow. It is extremaly important to provide as simplest *API Reference* as possible.

According to OWASP, *broken authentication* is one of most commonly security issue exposed in web applications.

Having this on mind we tried to build small set of necessary functions and created a lot of testing scenarios. Nevertheless, if anyone would found a bug in this application, please take a minute and file it at *issue-tracker*. Moreover, if someone would spot a *security hole* (a bug that might affect security of systems that use django-guardian as permission management library), please **DO NOT** create a public issue but contact me directly ([lukaszbalcerzak@gmail.com](mailto:lukaszbalcerzak@gmail.com)).

### 7.2.2 Running tests

Tests are run by Django's buildin test runner. To call it simply run:

```
$ python setup.py test
```

or inside a project with guardian set at INSTALLED\_APPS:

```
$ python manage.py test guardian
```

or using the bundled testapp project:

```
$ python manage.py test
```

### 7.2.3 Coverage support

*Coverage* is a tool for measuring code coverage of Python programs. It is great for tests and we use it as a backup - we try to cover 100% of the code used by django-guardian. This of course does *NOT* mean that if all of the codebase is covered by tests we can be sure there is no bug (as specification of almost all applications requiries some unique scenarios to be tested). On the other hand it definitely helps to track missing parts.

To run tests with `coverage` support and show the report after we have provided simple bash script which can be called by running:

```
$ ./run_test_and_report.sh
```

Result should be somehow similar to following:

```
(...)
.....
-----
Ran 48 tests in 2.516s

OK
Destroying test database 'default'...
Name                               Stmts   Exec  Cover   Missing
-----
guardian/__init__                   4        4   100%
guardian/backends                   20       20   100%
guardian/conf/__init__              1         1   100%
guardian/core                       29       29   100%
guardian/exceptions                  8         8   100%
guardian/management/__init__       10        10   100%
guardian/managers                   40       40   100%
guardian/models                     36       36   100%
guardian/shortcuts                  30       30   100%
guardian/templatetags/__init__      1         1   100%
guardian/templatetags/guardian_tags 39       39   100%
guardian/utils                      13       13   100%
-----
TOTAL                               231     231   100%
```

## 7.2.4 Tox

New in version 1.0.4.

We also started using `tox` to ensure `django-guardian`'s tests would pass on all supported Python and Django versions (see *Supported versions*). To use it, simply install `tox`:

```
pip install tox
```

and run it within `django-guardian` checkout directory:

```
tox
```

First time should take some time (it needs to create separate virtual environments and pull dependencies) but would ensure everything is fine.

## 7.2.5 Travis CI

New in version 1.0.4. Recently we have added support for `Travis`, continuous integration server so it is even more easy to follow if test fails with new commits: <http://travis-ci.org/#!/lukaszbdjango-guardian>.

## 7.3 Supported versions

`django-guardian` supports Python 2.7+/3.3+ and Django 1.7+.

### 7.3.1 Rules

- We would support Python 2.7. We also support Python 3.3+.
- Support for Python 3.3 may get dropped in the future.
- We support Django 1.7+. This is due to many simplifications in code we could do.

## 7.4 Changelog

### 7.4.1 Release 1.4.5 (9th August, 2016)

- Fix caching issue with `prefetch_perms`.
- Convert `readthedocs` link for their `.org` -> `.io` migration for hosted projects
- Remove `.permission_object` in `PermissionRequiredMixin`
- Add `PermissionListMixin` class
- Add example CRUD CBV project
- Add `TEMPLATES` in `example_project` settings
- Add `Queryset` support to `assign_perm`
- Added `QuerySet` support to `remove_perm`
- Updated `assign_perm` and `remove_perm` docstrings
- Moved `queryset` support in `assign_perms` to its own function
- Moved `queryset` support in `remove_perms` to its own function
- Consolidated `{User,Group}ObjectPermissionManager`, move logic of `bulk_*_perm` to managers
- `assign_perm` and `remove_perm` shortcuts accept `Permission` instance as `perm` and `QuerySet` as `obj` too.
- Consolidate `bulk_assign_perm` to `assign_perm` and `bulk_remove_perm` to `remove_perm`
- Upgraded Grappelli templates breadcrumbs block to new Django 1.9 and Grappelli 2.8 standards, including proper URLs and support for `preserved_filters`. Removed the duplicated `field.errors` in the `field.html` template file.
- Make `UserManage/GroupManage` forms overridable
- Fixed `GuardedModelAdminMixin` views render for Django 1.10

### 7.4.2 Release 1.4.4 (Apr 04, 2016)

- Don't install support `example_project`.
- Direct `ForeignKey` perms in `prefetch_perms`.

### 7.4.3 Release 1.4.3 (Apr 03, 2016)

- `guardian.VERSION` should be a tuple, not a list. Fixes #411.
- Support for prefetching permissions.
- Fixed union between queries.



- Allow specifying an empty list of permissions for `get_objects_for_group`.
- Mixed group and user direction relations broken. Fixes #271.
- Lookup anonymous user using custom username field.
- Fix up processing of `ANONYMOUS_USER_NAME` where set to `None`. Fixes #409.
- Require `TEMPLATE_403` to exist if `RENDER_403` set.

#### 7.4.4 Release 1.4.2 (Mar 09, 2016)

- Test against django-master (Django 1.10 - not released).
- Django 1.10 fixes.
- Fixes for documentation.
- PEP8 fixes.
- Fix distributed files in `MANIFEST.in`
- Use `pytest` for tests.
- Add dependancy on `django-envron`.
- Don't use `ANONYMOUS_USER_ID`. Uses `ANONYMOUS_DEFAULT_USERNAME` and `USER-NAME_FIELD` instead.
- Use `setuptools_scm` for versioning.
- Initialise admin context using `each_context` for Django  $\geq$  1.8.
- Add missing `with_superuser` parameter to `get_users_with_perms()`.
- Use `setuptools_scm` for versioning.
- Fixes for `example_project`.
- Only display permissions if permission actually assigned.
- When using `attach_perms` with `get_users_with_perms`, and `with_group_users` and `with_superuser` set to `False`, only directly assigned permissions are now returned, and not effective (inferred) permissions.

#### 7.4.5 Release 1.4.1 (Jan 10, 2016)

- Fix broken documentation.
- Fix `setup.py` errors (#387).
- Fix tox tests.
- Fix travis tests.

#### 7.4.6 Release 1.4.0 (Jan 8, 2016)

- Drop support for Django  $<$  1.7
- Drop support for django south migrations.
- Remove deprecated code.
- Fix many Django deprecated warnings.

- Fix tests and example\_project.
- Work around for postgresql specific Django bug (#366). This is a regression that was introduced in version 1.3.2.
- Updates to documentation.
- Require can\_change permission to change object perms in admin.
- Fixes broke admin URLs (#376 and #381).
- Tests now work with Mysql and Postgresql as well as sqlite.
- Uses django-envron for tests.

#### 7.4.7 Release 1.3.2 (Nov 14, 2015)

- Fixes tests for all versions of Django.
- Tests pass for Django 1.9b1.
- Drops support for Django < 1.5
- Add Russian translation.
- Various bug fixes.
- Ensure password for anonymous user is set to unusable, not None.

#### 7.4.8 Release 1.3.1 (Oct 20, 2015)

- Fixes for 1.8 compat

#### 7.4.9 Release 1.3 (Jun 3, 2015)

- Official Django 1.8 support (thanks to multiple contributors)

#### 7.4.10 Release 1.2.5 (Dec 28, 2014)

- Official Django 1.7 support (thanks Troy Grosfield and Brian May)
- Allow to override `PermissionRequiredMixin.get_permission_object`, part of `PermissionRequiredMixin.check_permissions` method, responsible for retrieving single object (Thanks zauddelig)
- French translations (Thanks Morgan Aubert)
- Added support for `User.get_all_permissions` (thanks Michael Drescher)

#### 7.4.11 Release 1.2.4 (Jul 14, 2014)

- Fixed another issue with custom primary keys at admin extensions (Thanks Omer Katz)

#### 7.4.12 Release 1.2.3 (Jul 14, 2014)

Unfortunately this was broken release not including any important changes.

### 7.4.13 Release 1.2.2 (Jul 2, 2014)

- Fixed issue with custom primary keys at admin extensions (Thanks Omer Katz)
- `get_403_or_None` now accepts Python path to the view function, for example `'django.contrib.auth.views.login'` (Thanks Warren Volz)
- Added `with_superuser` flag to `guardian.shortcuts.get_objects_for_user` (Thanks Bruno Ribeiro da Silva)
- Added possibility to disable monkey patching of the `User` model. (Thanks Cezar Jenkins)

### 7.4.14 Release 1.2 (Mar 7, 2014)

- Removed `get_for_object` methods from managers (#188)
- Extended documentation
- `GuardedModelAdmin` has been splitted into mixins
- Faster queries in `get_objects_for_user` when `use_groups=False` or `any_perm=True` (#148)
- Improved speed of `get_objects_for_user` shortcut
- Support for custom `User` model with not default username field
- Added `GUARDIAN_GET_INIT_ANONYMOUS_USER` setting (#179)
- Added `accept_global_perms` to `PermissionRequiredMixin`
- Added brazilian portuguese translations
- Added polish translations
- Added `wheel` support
- Fixed wrong anonymous user checks
- Support for Django 1.6
- Support for Django 1.7 alpha

---

**Important:** In this release we have removed undocumented `get_for_object` method from both `UserObjectPermissionManager` and `GroupObjectPermissionManager`. Not deprecated, removed. Those methods were not used within `django-guardian` and their odd names could lead to issues if user would believe they would return object level permissions associated with `user/group` and `object` passed as the input. If you depend on those methods, you'd need to stick with version 1.1 and make sure you do not misuse them.

---

### 7.4.15 Release 1.1 (May 26, 2013)

- Support for Django 1.5 (including Python 3 combination)
- Support for custom user models (introduced by Django 1.5)
- Ability to create permissions using Foreign Keys
- Added `user_can_access_owned_by_group_objects_only` option to `GuardedModelAdmin`.
- Minor documentation fixups
- Spanish translations

- Better support for `grappelli`
- Updated examples project
- Speed up `get_perms` shortcut function

#### 7.4.16 Release 1.0.4 (Jul 15, 2012)

- Added `GUARDIAN_RENDER_403` and `GUARDIAN_RAISE_403` settings (#40)
- Updated docstring for `get_obj_perms` (#43)
- Updated codes to run with newest `django-grappelli` (#51)
- Fixed problem with building a RPM package (#50)
- Updated caveats docs related with orphaned object permissions (#47)
- Updated `permission_required` docstring (#49)
- Added `accept_global_perms` for decorators (#49)
- Fixed problem with MySQL and booleans (#56)
- Added flag to check for *any* permission in `get_objects_for_user` and `get_objects_for_group` (#65)
- Added missing *tag closing* at template (#63)
- Added view mixins related with authorization and authentication (#73)
- Added `tox` support
- Added Travis support

#### 7.4.17 Release 1.0.3 (Jul 25, 2011)

- Added `get_objects_for_group` shortcut (thanks to Rafael Ponieman)
- Added `user_can_access_owned_objects_only` flag to `GuardedModelAdmin`
- Updated and fixed issues with example app (thanks to Bojan Mihelac)
- Minor typo fixed at documentation
- Included ADC theme for documentation

#### 7.4.18 Release 1.0.2 (Apr 12, 2011)

- `get_users_with_perms` now accepts `with_group_users` flag
- Fixed `group_id` issue at admin templates
- Small fix for documentation building process
- It's 2011 (updated dates within this file)

#### 7.4.19 Release 1.0.1 (Mar 25, 2011)

- `get_users_with_perms` now accepts `with_superuserusers` flag
- Small fix for documentation building process

#### 7.4.20 Release 1.0.0 (Jan 27, 2011)

- A final v1.0 release!

#### 7.4.21 Release 1.0.0.beta2 (Jan 14, 2011)

- Added `get_objects_for_user` shortcut function
- Added few tests
- Fixed issues related with `django.contrib.auth` tests
- Removed example project from source distribution

#### 7.4.22 Release 1.0.0.beta1 (Jan 11, 2011)

- Simplified example project
- Fixed issues related with test suite
- Added ability to clear orphaned object permissions
- Added `clean_orphan_obj_perms` management command
- Documentation cleanup
- Added `grappelli` admin templates

#### 7.4.23 Release 1.0.0.alpha2 (Dec 2, 2010)

- Added possibility to operate with global permissions for `assign` and `remove_perm` shortcut functions
- Added possibility to generate PDF documentation
- Fixed some tests

#### 7.4.24 Release 1.0.0.alpha1 (Nov 23, 2010)

- Fixed admin templates not included in `MANIFEST.in`
- Fixed admin integration codes

#### 7.4.25 Release 1.0.0.pre (Nov 23, 2010)

- Added admin integration
- Added reusable forms for object permissions management

### 7.4.26 Release 0.2.3 (Nov 17, 2010)

- Added `guardian.shortcuts.get_users_with_perms` function
- Added AUTHORS file

### 7.4.27 Release 0.2.2 (Oct 19, 2010)

- Fixed migrations order (thanks to Daniel Rech)

### 7.4.28 Release 0.2.1 (Oct 3, 2010)

- Fixed migration (it wasn't actually updating `object_pk` field)

### 7.4.29 Release 0.2.0 (Oct 3, 2010)

#### Fixes

- #4: guardian now supports models with not-integer primary keys and they don't need to be called "id".

---

**Important:** For 0.1.X users: it is required to *migrate* guardian in your projects. Add `south` to `INSTALLED_APPS` and run:

```
python manage.py syncdb
python manage.py migrate guardian 0001 --fake
python manage.py migrate guardian
```

---

#### Improvements

- Added `South` migrations support

### 7.4.30 Release 0.1.1 (Sep 27, 2010)

#### Improvements

- Added view decorators: `permission_required` and `permission_required_403`

### 7.4.31 Release 0.1.0 (Jun 6, 2010)

- Initial public release

---

**License**

---

Copyright (c) 2010-2014 Lukasz Balcerzak <lukaszbalcerzak@gmail.com>  
All rights reserved.

Redistribution and use in source and binary forms, with or without  
modification, are permitted provided that the following conditions are met:

- \* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- \* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The SVG icons in guardian/static/guardian/img are copied from Django.

SVG icons source: <https://github.com/encharm/Font-Awesome-SVG-PNG>  
Font-Awesome-SVG-PNG is licensed under the MIT license:

The MIT License (MIT)

Copyright (c) 2014 Code Charm Ltd

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.



---

## Indices and tables

---

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



## g

`guardian.admin`, 25  
`guardian.backends`, 26  
`guardian.core`, 27  
`guardian.decorators`, 27  
`guardian.forms`, 29  
`guardian.managers`, 30  
`guardian.mixins`, 31  
`guardian.models`, 33  
`guardian.shortcuts`, 33  
`guardian.templatetags.guardian_tags`, 39  
`guardian.utils`, 38



**A**

admin

GuardedModelAdmin, 25

ANONYMOUS\_USER\_NAME

setting, 9

are\_obj\_perms\_required()

(guardian.forms.BaseObjectPermissionsForm method), 30

assign\_perm() (in module guardian.shortcuts), 33

**B**

BaseObjectPermission

model, 33

BaseObjectPermission (class in guardian.models), 33

BaseObjectPermissionsForm

form, 29

BaseObjectPermissionsForm (class in guardian.forms), 30

**C**

check\_permissions() (guardian.mixins.PermissionRequiredMixin method), 32

clean\_orphan\_obj\_perms

command, 30

clean\_orphan\_obj\_perms() (in module guardian.utils), 39

command

clean\_orphan\_obj\_perms, 30

Command (class in guardian.management.commands.clean\_orphan\_obj\_perms),

30

**F**

form

BaseObjectPermissionsForm, 29

GroupObjectPermissionsForm, 29

UserObjectPermissionsForm, 29

**G**

get\_all\_permissions() (guardian.backends.ObjectPermissionBackend method), 26

get\_anonymous\_user() (in module guardian.utils), 39

get\_group\_perms() (in module guardian.shortcuts), 34

get\_groups\_with\_perms() (in module guardian.shortcuts), 35

get\_identity() (in module guardian.utils), 39

get\_local\_cache\_key() (guardian.core.ObjectPermissionChecker method), 27

get\_obj\_perms() (in module guardian.templatetags.guardian\_tags), 40

get\_obj\_perms\_field() (guardian.forms.BaseObjectPermissionsForm method), 30

get\_obj\_perms\_field\_choices() (guardian.forms.BaseObjectPermissionsForm method), 30

get\_obj\_perms\_field\_class() (guardian.forms.BaseObjectPermissionsForm method), 30

get\_obj\_perms\_field\_initial() (guardian.forms.BaseObjectPermissionsForm method), 30

get\_obj\_perms\_field\_label() (guardian.forms.BaseObjectPermissionsForm method), 30

get\_obj\_perms\_field\_name() (guardian.forms.BaseObjectPermissionsForm method), 30

get\_obj\_perms\_field\_widget() (guardian.forms.BaseObjectPermissionsForm method), 30

get\_objects\_for\_group() (in module guardian.shortcuts), 37

get\_objects\_for\_user shortcut, 36

get\_objects\_for\_user() (in module guardian.shortcuts), 36

get\_perms() (guardian.core.ObjectPermissionChecker method), 27

get\_perms() (in module guardian.shortcuts), 34

get\_perms\_for\_model() (in module guardian.shortcuts), 34

get\_required\_permissions() (guardian.mixins.PermissionRequiredMixin method), 32

- get\_user\_perms() (in module guardian.shortcuts), 34
- get\_users\_with\_perms() (in module guardian.shortcuts), 35
- GroupObjectPermission
  - model, 33
- GroupObjectPermission (class in guardian.models), 33
- GroupObjectPermissionManager
  - manager, 30
- GroupObjectPermissionManager (class in guardian.managers), 31
- GroupObjectPermissionsForm
  - form, 29
- GroupObjectPermissionsForm (class in guardian.forms), 29
- GuardedModelAdmin
  - admin, 25
- GuardedModelAdmin (class in guardian.admin), 25
- guardian.admin (module), 25
- guardian.backends (module), 26
- guardian.core (module), 27
- guardian.decorators (module), 27
- guardian.forms (module), 29
- guardian.managers (module), 30
- guardian.mixins (module), 31
- guardian.models (module), 33
- guardian.shortcuts (module), 33
- guardian.templatetags.guardian\_tags (module), 39
- guardian.utils (module), 38
- GUARDIAN\_GET\_INIT\_ANONYMOUS\_USER
  - setting, 10
- GUARDIAN\_RAISE\_403
  - setting, 9
- GUARDIAN\_RENDER\_403
  - setting, 9
- GUARDIAN\_TEMPLATE\_403
  - setting, 9
- H**
- has\_perm() (guardian.backends.ObjectPermissionBackend method), 26
- has\_perm() (guardian.core.ObjectPermissionChecker method), 27
- L**
- LoginRequiredMixin
  - mixin, 31
- LoginRequiredMixin (class in guardian.mixins), 31
- M**
- manager
  - GroupObjectPermissionManager, 30
  - UserObjectPermissionManager, 30
- mixin
  - LoginRequiredMixin, 31
  - PermissionRequiredMixin, 31
- model
  - BaseObjectPermission, 33
  - GroupObjectPermission, 33
  - UserObjectPermission, 33
- O**
- ObjectPermissionBackend (class in guardian.backends), 26
- ObjectPermissionChecker (class in guardian.core), 27
  - on\_permission\_check\_fail() (guardian.mixins.PermissionRequiredMixin method), 32
- P**
- permission\_required() (in module guardian.decorators), 27
- permission\_required\_or\_403() (in module guardian.decorators), 28
- PermissionRequiredMixin
  - mixin, 31
- PermissionRequiredMixin (class in guardian.mixins), 31
- prefetch\_perms() (guardian.core.ObjectPermissionChecker method), 27
- R**
- remove\_perm() (in module guardian.shortcuts), 34
- S**
- save\_obj\_perms() (guardian.forms.BaseObjectPermissionsForm method), 30
- save\_obj\_perms() (guardian.forms.GroupObjectPermissionsForm method), 29
- save\_obj\_perms() (guardian.forms.UserObjectPermissionsForm method), 29
- setting
  - ANONYMOUS\_USER\_NAME, 9
  - GUARDIAN\_GET\_INIT\_ANONYMOUS\_USER, 10
  - GUARDIAN\_RAISE\_403, 9
  - GUARDIAN\_RENDER\_403, 9
  - GUARDIAN\_TEMPLATE\_403, 9
- shortcut
  - get\_objects\_for\_user, 36
- U**
- UserObjectPermission
  - model, 33
- UserObjectPermission (class in guardian.models), 33
- UserObjectPermissionManager
  - manager, 30
- UserObjectPermissionManager (class in guardian.managers), 30

UserObjectPermissionsForm  
form, 29

UserObjectPermissionsForm (class in guardian.forms),  
29