guardian Documentation

Release 1.2.5

Lukasz Balcerzak

November 12, 2015

Date November 12, 2015

Version 1.2.5

Documentation:

CHAPTER 1

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 [1] 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.

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

CHAPTER 2

Installation

This application requires Django 1.3 or higher and it is 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 boundled example application, there are some other requirements. See more details about the topics:

- *Testing*
- *Example project*

CHAPTER 3

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',
)
```

As django-guardian supports anonymous user's object permissions we also need to add following to our settings module:

```
ANONYMOUS_USER_ID = -1
```

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).

If ANONYMOUS_USER_ID 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.

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

CHAPTER 4

Optional settings

In addition to required `ANONYMOUS_USER_ID` setting, 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_DEFAULT_USERNAME_VALUE

New in version 1.1.

Due to changes introduced by Django 1.5 user model can have differently named `username` field (it can be removed too, but `guardian` currently depends on it). After `syncdb` command we create anonymous user for convenience, however it might be necessary to set this configuration in order to set proper value at `username` field.

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

CHAPTER 5

User Guide

5.1 Assign object permissions

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

5.1.1 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` 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 http://docs.djangoproject.com/en/1.2/topics/auth/#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*.

5.1.2 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
```

5.2 Check object permissions

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

5.2.1 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`.

5.2.2 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.

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.

5.2.3 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

5.3 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).

5.3.1 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
```

5.4 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.

5.4.1 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 example_project.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 example_project.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 thats 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.

5.5 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.

5.5.1 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'
```

5.6 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.

5.6.1 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.

5.6.2 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 Caveats

5.7.1 Orphaned object permissions

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, permisions 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:

```python
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*

CHAPTER 6

API Reference

6.1 Admin

6.1.1 GuardedModelAdmin

6.2 Backends

6.2.1 ObjectPermissionBackend

6.3 Core

6.3.1 ObjectPermissionChecker

6.4 Decorators

6.4.1 permission_required

6.4.2 permission_required_or_403

6.5 Forms

6.5.1 UserObjectPermissionsForm

6.5.2 GroupObjectPermissionsForm

6.5.3 BaseObjectPermissionsForm

6.6 Management commands

6.7 Managers

6.7.1 UserObjectPermissionManager

6.7.2 GroupObjectPermissionManager

6.8 Mixins

6.8.1 LoginRequiredMixin

6.8.2 PermissionRequiredMixin

6.9 Models

6.9.1 BaseObjectPermission

6.9.2 UserObjectPermission

6.9.3 GroupObjectPermission

6.10 Shortcuts

6.10.1 assign_perm

6.10.2 remove_perm

6.10.3 get_perms

6.10.4 get_perms_for_model

6.10.5 get_users_with_perms

6.10.6 get_groups_with_perms

6.10.7 get_objects_for_user

6.10.8 get_objects_for_group

6.11 Utilities

6.11.1 get_anonymous_user

6.11.2 get_identity

6.11.3 clean_orphan_obj_perms

6.12 Template tags

6.12.1 get_obj_perms

CHAPTER 7

Development

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/lukaszb/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/lukaszb/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/lukaszb/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.

5. (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.

6. Publish changes:

```
$ git push origin YOUR_BRANCH_NAME
```

6. 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 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 explonation 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 Example project

Example project should be boundled 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:

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

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

```
$ python manage.py syncdb
```

Finally we can run dev server:

```
$ python manage.py runserver
```

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

Note: Example project must be run with Django 1.5 or later. This is to ensure that custom user model can be used.

7.3 Testing

7.3.1 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. Neverteless, 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).

7.3.2 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
```

7.3.3 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 requries 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 by 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%
```

7.3.4 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.

7.3.5 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/#!/lukaszb/django-guardian.

7.4 Supported versions

`django-guardian` supports Python 2.6+/3.3+ and Django 1.3+. Also, we support `django-grappelli` 2.3.5.

Note: `django-grappelli` support was an experiment. We are not going to maintain this support starting with version 1.2. It means that since v1.2 grappelli parts within guardian would be deprecated. And removed before next big release (which most probably would be 2.0).

7.4.1 Rules

- We would support Python 2.7 and Python 2.6 (until Django drops support for 2.6). Since Django 1.5 we also support Python 3.3+.
- We support Django 1.3+, however next big `guardian` release (v2.0) we would support Django 1.6+ (or higher, depending on the date guardian v2.0 would be released). This is due to many simplifications in code we could do. There is also a chance that v2.0 would drop support for Python 2.X.

7.5 Changelog

7.5.1 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)

7.5.2 Release 1.2.4 (Jul 14, 2014)

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

7.5.3 Release 1.2.3 (Jul 14, 2014)

Unfortunately this was broken release not including any important changes.

7.5.4 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)

7.5.5 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.

7.5.6 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

7.5.7 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 oprhaned 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

7.5.8 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

7.5.9 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)

7.5.10 Release 1.0.1 (Mar 25, 2011)

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

7.5.11 Release 1.0.0 (Jan 27, 2011)

- A final v1.0 release!

7.5.12 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

7.5.13 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

7.5.14 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

7.5.15 Release 1.0.0.alpha1 (Nov 23, 2010)

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

7.5.16 Release 1.0.0.pre (Nov 23, 2010)

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

7.5.17 Release 0.2.3 (Nov 17, 2010)

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

7.5.18 Release 0.2.2 (Oct 19, 2010)

- Fixed migrations order (thanks to Daniel Rech)

7.5.19 Release 0.2.1 (Oct 3, 2010)

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

7.5.20 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

7.5.21 Release 0.1.1 (Sep 27, 2010)

Improvements

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

7.5.22 Release 0.1.0 (Jun 6, 2010)

- Initial public release

CHAPTER 8

License

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Index