django-bop Documentation

Release 0.1

Peter van Kampen

July 01, 2013

CONTENTS

1 Installation **3**

2 Grant and Revoke **5**
- 2.1 ObjectAdmin . . . 5
- 2.2 Form Factory . . . 5
- 2.3 API . . . 6

3 Checking **7**
- 3.1 ObjectBackend . . . 7
- 3.2 Decorator . . . 7
- 3.3 TemplateTag . . . 8
- 3.4 ObjectPermissionManager . . . 8
- 3.5 UserObjectManager . . . 8
- 3.6 has_model_perms . . . 9

django-bop provides tools to manage object-level permissions.

Contents:

CHAPTER

ONE

INSTALLATION

Install django-bop in your (virtual) environment:

```
$ pip install django-bop
```

If you haven't already you should also install south:

```
$ pip install South
```

Add 'bop' (and south) to you INSTALLED_APPS in settings.py:

```
INSTALLED_APPS = (
  ...
  'south',
  'bop',
)
```

While in settings.py specify the AUTHENTICATION_BACKENDS:

```
AUTHENTICATION_BACKENDS = (
      'django.contrib.auth.backends.ModelBackend',
      'bop.backends.ObjectBackend',
)
```

If you, optionally, want to give permissions to anonymous users you should do the following:

1. Add a user to contrib.auth.models.User to represent anonymous users (e.g. via the admin). Give it an appropriate name (anon / anonymous) so it easily recognized when assigning permissions.
2. Add ANONYMOUS_USER_ID to settings.py:

   ```
   ANONYMOUS_USER_ID = 2
   ```

If, in addition – and again optionally – you want to support Model-permissions for anonymous users, you can add the AnonymousModelBackend:

```
AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.ModelBackend',
    'bop.backends.AnonymousModelBackend',
    'bop.backends.ObjectBackend',
)
```

When all configuration is done, bring the database up to date:

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

CHAPTER

TWO

GRANT AND REVOKE

Once installed you will want to start granting permissions to your objects. django-bop provides three tools to help you with that:

- *ObjectAdmin*
- *Form Factory*
- *API*

2.1 ObjectAdmin

By subclassing ObjectAdmin in admin.py (in stead of ModelAdmin) you can manage the objects in the django admin. Each object (detail page) will have inline forms to grant / revoke (delete) permissions.

The admin will also filter out objects that the user doesn't have acces to or deny actions he/she doesn't have permissions for.

In admin.py:

```
from django.contrib import admin

from bop.admin import ObjectAdmin

from myapp.models import MyModel


class MyModelAdmin(ObjectAdmin):
    # All the usual options work here
    pass


admin.site.register(MyModel, MyModelAdmin)
```

2.2 Form Factory

If you want to have the inlines in your own (Model)forms you can use the inline_permissions_form_factory to generate a formset that will handle the permissions for you:

```
# TODO Example
from bop.forms import inline_permissions_form_factory
```

2.3 API

Bop provides two very flexible functions to grant and revoke permissions for objects to users and groups:

```
from bop.api import grant, revoke

grant([mymodeladmin, testuser], None, 'myapp.delete_mymodel', MyModel.objects.filter(owner=testuser))

revoke(testuser, None, 'myapp.delete_mymodel', MyModel.objects.filter(id=1))
```

Both functions have the same signature: users, groups, permissions, objects. All arguments can be a single item or an iterable.:

```
grant(User.objects.all(), Group.objects.all(), 'myapp.can_view', MyModel.objects.all())
```

Users, groups and permissions can be either a string or an object.

- For users a string, or iterable of strings, will be converted into a User object by doing User.objects.get(username=user)
- For groups a string will be converted into a Group object by doing Group.objects.get(name=group)
- For permisions a string will be converted into a Permission object by doing (simplefied here) Permission.objects.get(app_label=app_label, codename=codename)

Objects however must be instances of a model that is 'registered' / known in django.contrib.contenttyes.

CHAPTER

THREE

CHECKING

Once permissions have been granted to objects you want to check these permissions in your views and templates. Bop provides several mechanisms to do that.

- *ObjectBackend*
- *Decorator*
- *TemplateTag*
- *ObjectPermissionManager*
- *UserObjectManager*
- *has_model_perms*

3.1 ObjectBackend

Provided you installed bop per the *instructions* you can use the standard django method of checking for permissions in your views:

```
testuser.has_perm('myapp.delete_mymodel', myobject)
testuser.get_all_permissions(myobject)
testuser.get_group_permissions(myobject)
```

3.2 Decorator

The `user_has_object_level_perm` decorator checks wether a user has permission to access an object:

```
:py:obj:`user_has_object_level_perm`(perm, model, pkfield='pk', login_url=None, redirect_field_name=
```

The `pkfield` is expected to be passed to the view as a keyword argument.

The object will be obtained by doing:

```
Model.objects.get(**{pkfield:kwargs[pkfield]})
```

An example will perhaps better illustrate:

```
# In urls.py
...
(r'^articles/(\d{4})/(\d{2})/(?P<article_id>\d+)/$', 'news.views.article_detail'),
...
```

```
# In views.py
from bop.decorators import user_has_object_level_perm

from news.models import Article


@user_has_object_level_perm('news.view_article', Article, pkfield='article_id')
def view article_detail(year, month, article_id):
    pass
```

Note that the `pkfield` must be using *named groups* so the decorator can actually find the keyword argument in **kwargs.

3.3 TemplateTag

Bop borrowed code from django-authority to provide a templatetag `ifhasperm`:

```
{% load permissions %}

{% ifhasperm PERMISSION_LABEL USER OBJ %}
    lalala
{% else %}
    meh
{% endifhasperm %}

{% ifhasperm "change_poll" request.user poll %}
    lalala
{% else %}
    meh
{% endifhasperm %}
```

3.4 ObjectPermissionManager

The objectpermissionmanager has three methods to query the ObjectPermissions granted to users:

- `get_for_model(model)`

  returns all ObjectPermissions for the given model

- `get_for_user(user)`

  returns all ObjectPermissions for the given user

- `get_for_model_and_user(model, user)`

  returns all ObjectPermissions for the given model and user

3.5 UserObjectManager

The UserObjectManager can be added to any Model and it will work like the default manager with one extra method:

- `get_user_objects(user, permissions=None, check_model_perms=False)`

Will only return objects the given user has permissions on and optionally filter for specific permissions.

You can use the manager on any model:

```
from bop.managers import UserObjectManager

class MyModel(models.Model):
    name = models.CharField(max_length=255)
    ...

    objects = UserObjectManager()
```

And it will work like the normal manager but rather than getting all objects and checking the permissions in the template you can filter the objects this user has permissions for:

```
# This will return all objects for which a permission has been
# granted to testuser
MyModel.objects.get_for_user(testuser)

# This will return all objects for which a *specific* permission has
# been granted to testuser
MyModel.objects.get_for_user(testuser, permissions=['myapp.can_view'])
```

When both model- and objectpermission have been granted the manager will, by default, only check the objectpermissions. You can override that by setting the check_model_perms to `True`.

3.6 has_model_perms

It is possible that some permission was granted to one model in a module but not to another model in the same application. When `get_for_user` is called with `check_model_perms=True` bop checks the permissions for the *model*, not the *module* by calling `bop.api.has_model_perms(user, model)`.