privileges Documentation

Release 0.1

Eldarion, Inc

March 04, 2013

CONTENTS

Unlike Django permissions, `privileges` is not tied to operations on individual models. It operates at a higher level of abstraction and is instead concerned more with providing the site developer complete freedom in determining who can do what. There certainly is some overlap with the built in permissions system, and while you could use `privileges` to replace it, at least large parts of it, that is not the aim of this app.

Instead, think of `privileges` allowing the site developer to control access to certain features. Operating at the template and view layers the site developer can paint as broad or as fine of strokes to suit their needs.

It is extensible in the sense that the site developer can define and register their own privilege validation handlers. In fact they must define at least one handler. There is a template tag for checking privileges in templates and a decorator for checking privileges when a view is called.

There is a model that stores the named privileges which are nothing more than named slugs. The records carry no special meeting to `privileges` in isolation but depend on the site developer to impart meaning through reference in his site.

CHAPTER

ONE

DEVELOPMENT

The source repository can be found at https://github.com/eldarion/privileges

CHAPTER

TWO

SPONSORSHIP

development sponsored by Midwest Communications

2.1 Contents

2.1.1 ChangeLog

0.1

- initial release

2.1.2 Installation

- To install

```
pip install privileges
```

- Add 'privileges' to your INSTALLED_APPS setting:

```
INSTALLED_APPS = (
    # other apps
    "privileges",
)
```

After adding to your settings.INSTALLED_APPS, you will need to add the privileges you plan on using throughout your site. It is best to create them using the /admin/ and then as you make changes or add new ones, update your fixtures/initial_data.json fixture:

```
./manage.py dumpdata privileges --indent=4
```

Capture the output and merge it into your initial_data.json.

2.1.3 Template Tags

In order to assist in validating privileges in the template to control bits of your UI, there is a template tag called check_privilege and it is used like so:

```
{% load privileges_tags %}
....
{% check_privilege 'foo_feature_enabled' for user as has_foo %}

{% if has_foo %}
    ....
{% endif %}
```

2.1.4 Decorators

While the template tag is good to control bits in the UI, you will likely want to make sure POST requests can't be forged. Just because you don't show a form in the UI, doesn't mean there isn't a url accepting POST requests. This is the reason for the `privilege_required` decorator.

By putting this decorator on views, it will validate that the user calling the view as the specified privilege, otherwise it will redirect, by default, to the login url:

```
from privileges.decorators import privilege_required


@privilege_required("widget_management_feature_enabled")
def add_widget(request):
    ....
```

2.1.5 Grants

Privileges can be granted to other users and who is allowed to grant what to whom can be controlled via the implementation of a couple site level callables. It defaults to a wide-open system. In other words, no restrictions on anyone granting any of their privileges to any other user in the site.

Grants are an entirely optional feature. Simply don't add the urls and the feature will be inaccessible to users.

Installation

To add grants to your site, you are essnetially just exposing the UI to your users to be able to create and manage their grants. The simplest form of enabling granting is:

```
...
url(r"^privileges/", include("privileges.urls")),
...
```

This will add four urls to your url configuration:

- `privileges_grant_list`
- `privileges_grant_create`
- `privileges_grant_update`
- `privileges_grant_delete`

These all take `username` as a kwarg and the update and delete urls also take the `pk` of the grant objecct. You might want to link to this pages under an account settings interface for the user in your site somewhere.

privileges_grant_list

kwargs username

context grants_list, username

template `privileges/grant_list.html`

This view will display the user's grants and the requesting user has to either match the usenrame or be a superuser. It will render a template stored at `privileges/grant_list.html` and a default template that exiends `site_base.html` has been included in this package.

privileges_grant_create

kwargs username

context form, username

template `privileges/grant_form.html`

This view handles the form display and POST handling to create new grants.

privileges_grant_update

kwargs username, pk

context form, grant, username

template `privileges/grant_form.html`

This view handles the form display and POST handling to update existing grants.

privileges_grant_delete

kwargs username, pk

context form, grant, username

template `privileges/grant_confirm_delete.html`

This view handles the form display and POST handling to delete grants.

Customization

There are two callables that you can define in your site and configure via settings. They currently default to:

```
PRIVILEGES_PRIVILEGE_LIST_CALLABLE = "privileges.grants._privilege_list"
PRIVILEGES_GRANTEE_LIST_CALLABLE = "privileges.grants._grantee_list"
```

These should be callables that are importable within the context of your site. Futhermore, they are expected to have the following argspecs:

```
privilege_list(grantor, grantee=None)

grantee_list(grantor, privilege=None)
```

Where `grantor` and `grantee` are `auth.User` objects, and `privilege` is a `privileges.Privilege` object.

These functions are what control the options in the `privileges.forms.GrantForm` that validate and allow the creation of new grants by users of your site.

These functions currently return all privileges and all users (excluding only the `grantor` from the list), so it is wide open by default, and is up to you to implement the business rules for how these lists should be constrained.

2.1.6 Usage

The best way to familiarize yourself with `privileges` is to walk through some examples. So let's get started.

Profile Based Privileges

You are building a site that has a number of different personas so you decide to model that using `idios` and end up with something that looks like:

```
from idios.models import ProfileBase


class Persona(ProfileBase):

    name = models.CharField(max_length=50, null=True, blank=True)


class MemberPersona(Persona):

    expired = models.BooleanField(default=False)


class StaffPersona(Persona):

    pass
```

You will need to add and register a privileges handler:

```
from idios.models import ProfileBase
from privileges.models import Privilege
from privileges.registration import registry


class Persona(ProfileBase):

    name = models.CharField(max_length=50, null=True, blank=True)


class MemberPersona(Persona):

    expired = models.BooleanField(default=False)


class StaffPersona(Persona):

    pass
```

```
class PersonaPrivilege(models.Model):

    persona_type = models.ForeignKey(ContentType)
    privilege = models.ForeignKey(Privilege)

    class Meta:
        verbose_name = "Persona Privilege"
        unique_together = ["persona_type", "privilege"]

    def __unicode__(self):
        return unicode("%s has '%s'" % (self.persona_type, self.privilege.label))


def has_privilege(user, privilege):
    """
    Checks each Persona that a user has and it's privileges
    """
    if user.is_superuser:
        return True

    for p in [MemberPersona, StaffPersona]:
        for persona in p.objects.filter(user=user):
            ct_type = ContentType.objects.get_for_model(persona)
            if PersonaPrivilege.objects.filter(
                persona_type=ct_type,
                privilege__label=privilege
            ).exists():
                return True
    return False


registry.register(has_privilege)
```

As you can see above, I added `has_privilege` and registered it with `registry.register`.

The handler that you register can be any callable that takes two parameters, a user object, and a string that matches the label of one of the privilege objects in your database.

Achievement Based Privileges

Another example of how you might employ the use of `privileges` in your project is by only giving users that have earned a certain reputation or score depending on your chosen nomenclature. Using another open source app by Eldarion, `brabeion`, we can hook in the same type of handler.

First a quick setup of `braebion`. Start an a new app in your project. Let's call it `glue` as that's what it's doing – gluing parts of different apps together. So in `glue/badges.py` you will have:

```
from brabeion.base import Badge, BadgeAwarded


class ProfileCompletionBadge(Badge):
    slug = "profile_completion"
    levels = [
        "Bronze",
        "Silver",
        "Gold",
    ]
    events = [
```

```
        "profile_updated",
    ]
    multiple = False

    def award(self, **state):
        user = state["user"]
        profile = user.get_profile()

        if profile.name and profile.about and profile.location and profile.website:
            return BadgeAwarded(level=3)
        elif profile.name and profile.about and profile.location:
            return BadgeAwarded(level=2)
        elif profile.name and profile.location:
            return BadgeAwarded(level=1)
```

Then in `glue/models.py` will want to create a model to link the `ProfileCompletionBadge` with a certain set of privileges. In addition, we write and register the `has_privilege` handler here as well:

```
from django.db import models
from django.db.models.signals import post_save

from brabeion import badges

from glue.badges import ProfileCompletionBadge
from personas.models import DefaultPersona
from privileges.models import Privilege
from privileges.registration import registry


BADGE_CHOICES = [
    (
        "%s:%s" % (ProfileCompletionBadge.slug, x[0]),
        "%s - %s" % (ProfileCompletionBadge.slug, x[1])
    )
    for x in enumerate(ProfileCompletionBadge.levels)
]


class BadgePrivilege(models.Model):

    badge = models.CharField(max_length=128, choices=BADGE_CHOICES)
    privilege = models.ForeignKey(Privilege)


def has_privilege(user, privilege):
    if not hasattr(user, "badges_earned"):
        return False

    for b in user.badges_earned.all():
        badge = "%s:%s" % (b.slug, b.level)
        if BadgePrivilege.objects.filter(
            badge=badge,
            privilege__label__iexact=privilege
        ).exists():
            return True

    return False
```

```
def handle_saved_persona(sender, instance, created, **kwargs):
    badges.possibly_award_badge("profile_updated", user=instance.user)


badges.register(ProfileCompletionBadge)
post_save.connect(handle_saved_persona, sender=DefaultPersona)
registry.register(has_privilege)
```

As you will notice from the code above, the implementation of the handler is completely different from that of the Persona handler written about previously. Don't be distracted by the braebion details around badges and whatnot, the important thing to realize is that you, the site developer (or app developer), can control exactly how different privileges are evaluated in contexts that you control.

In addition, this example and the previous example where we attached privileges to personas/profiles, are not mutually exclusive. They can work together. What happens when privileges are checked is that all registered handlers are evaluated until either it either finds one that evaluates to True or gets to the end of all registered handlers, which it then will return False.