Bridgekeeper Documentation

Release 0.6.dev13+g30b07fa.d20190724

Leigh Brenecki, and contributors

Jul 24, 2019

Tutorial

> Who would cross the Bridge of Death
> must answer me these questions three,
> ere the other side he see.
>
> —The Bridgekeeper, *Monty Python and the Holy Grail*

Bridgekeeper is a permissions library for Django projects, where permissions are defined in your code, rather than in your database.

It's heavily inspired by django-rules, but with one important difference: **it works on QuerySets as well as individual model instances**.

This means that you can efficiently show a `ListView` of all of the model instances that your user is allowed to edit, for instance, without having your permission-checking code in two different places.

Bridgekeeper works on Django 2.0+ on Python 3.5+, and is licensed under the MIT License.

Warning: Bridgekeeper (and these docs!) are a work in progress.

CHAPTER 1

Installing Bridgekeeper

First, install the `bridgekeeper` package from PyPI.

```
$ pip install bridgekeeper
# or, if you're using pipenv
$ pipenv install bridgekeeper
```

Then, add Bridgekeeper to your `settings.py`:

```
 INSTALLED_APPS = (
     'django.contrib.admin',
     'django.contrib.auth',
     # ...
+    'bridgekeeper',
 )

 # ...

 AUTHENTICATION_BACKENDS = (
     'django.contrib.auth.backends.ModelBackend',
+    'bridgekeeper.backends.RulePermissionBackend',
 )
```

Note: Order doesn't matter for either the `INSTALLED_APPS` or `AUTHENTICATION_BACKENDS` entry.

You might not already have the `AUTHENTICATION_BACKENDS` setting in your `settings.py`; if not, you'll have to add it.

CHAPTER 2

Defining Permissions

In this tutorial, we'll be using a example app, an online stock management portal for shrubberies; we'll define some permissions for it in this section, then use them in views in the next section. It has a single app called `shrubberies`, with a `models.py` looks something like this:

Listing 2.1: shrubberies/models.py

```
from django.contrib.auth.models import User
from django.db import models


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


class Branch(models.Model):
    store = models.ForeignKey(Store, on_delete=models.CASCADE)
    name = models.CharField(max_length=255)


class Shrubbery(models.Model):
    branch = models.ForeignKey(Branch, on_delete=models.PROTECT)
    name = models.CharField(max_length=255)
    price = models.DecimalField(max_digits=5, decimal_places=2)


class Profile(models.Model):
    """User profile.

    Every user has one Profile object attached to them, which is
    automatically created when the user is added, and holds information
    about which branch of which store they belong to and what their
    role is.
    """

    user = models.OneToOneField(User, on_delete=models.CASCADE)
```

```
    branch = models.ForeignKey(Branch, on_delete=models.PROTECT)
    role = models.CharField(max_length=16, choices=(
        ('apprentice', 'Apprentice Shrubber'),
        ('shrubber', 'Shrubber'),
    ))
```

2.1 Defining Our First Permission

In Bridgekeeper, permissions are defined by **rules**. A rule is an object that can be given a user and a model instance, and decides whether or not to allow that user access to that instance.

Note: From that description, you might be thinking that a rule object is just a function with the signature `(user, model_instance) -> bool`. While you can certainly think of them that way, internally they're a little more complex than that, for reasons that will become apparent in the next section.

One of the simplest rules in Bridgekeeper is the built-in *`is_staff`* rule, which answers "yes" if the user trying to log in has `is_staff` set, or "no" otherwise.

We turn a rule into a **permission** by assigning it to a name. We do that by creating a file called `permissions.py` inside our app, importing `bridgekeeper.perms` (which is a Python dictionary[1] that maps permission names to their corresponding rules) and adding entries to it.

Listing 2.2: shrubberies/permissions.py

```
from bridgekeeper import perms
from bridgekeeper.rules import is_staff

perms['shrubbery.create_store'] = is_staff
perms['shrubbery.update_store'] = is_staff
perms['shrubbery.delete_store'] = is_staff
```

Note: We've used permission names that follow the convention set by Django's built-in permissions mechanism, so that they're used by other apps that expect that naming convention, such as Django's built-in admin. You can use whatever permission names you like, although it's best to namespace them with the name of your app followed by a full stop at the start (e.g. `shrubbery.foo`).

These permissions are now fully working; if you wanted, you could skip right through to the next section to see how to use them in your views. Don't, though, because Bridgekeeper is capable of far more.

2.2 Blanket Rules

A blanket rule is a rule that decides whether or not to allow access based solely on the user that's trying to access the resource. They'll either allow access to everything or nothing at all, hence the name.

We've already used one blanket rule—the built-in *`is_staff`* rule—but we can also define our own, by using the `blanket_rule` decorator to wrap a function that takes a user and returns a boolean.

[1] `bridgekeeper.perms` is actually an instance of `PermissionMap`, which is a subclass of `dict` with a few small changes, but you can treat it as a normal dictionary anyway.

In this example, we're using the `role` attribute on each user's associated `Profile` instance to restrict access to users that have been assigned a particular role:

Listing 2.3: shrubberies/rules.py

```
from bridgekeeper.rules import blanket_rule

@blanket_rule
def is_apprentice(user):
    return user.profile.role == 'apprentice'

@blanket_rule
def is_shrubber(user):
    return user.profile.role == 'shrubber'
```

If we were given a requirement like this:

> Only shrubbers can edit shrubberies.

We could use our new `is_shrubber` rule the same way that we used `is_staff` before:

Listing 2.4: shrubberies/permissions.py

```
from .rules import is_shrubber

perms['shrubbery.update_shrubbery'] = is_shrubber
```

2.3 Matching Against Model Instance Attributes

Blanket rules let us allow or deny access to entire model classes based on the user, but we can also allow access to only certain instances. Consider the following requirement:

> Users can only edit shrubberies that belong to their branch.

We can model this as a Bridgekeeper rule by creating an instance of the *`Attribute`* class:

Listing 2.5: shrubberies/permissions.py

```
from bridgekeeper.rules import Attribute

perms['shrubbery.update_shrubbery'] = Attribute('branch', lambda user: user.profile.
↪branch)
```

You can think of *`Attribute`* as the Bridgekeeper equivalent to the standard library's `getattr()` function. It will only allow access when the attribute named in the first argument (here, `'branch'`) matches whatever is in the second argument. The second argument can either be a constant, or—as we've used here—a function that takes the current user and returns something to match against.

2.4 Traversing Relationships

What if we change the requirement to something like this?

> Users can only edit shrubberies that belong to their store.

Shrubberies don't have a `store` attribute; we have to go through the `branch` attribute to figure out which store a shrubbery belongs to, so we can't use *`Attribute`*.

This is where the *Relation* class comes in. *Relation* is similar to *Attribute*, but instead of taking a constant or function as its last argument, it takes *another rule object*, which is applied to the other side of the relation.

Listing 2.6: shrubberies/permissions.py

```
from bridgekeeper.rules import Relation

from . import models

perms['shrubbery.update_shrubbery'] = Relation(
    'branch',
    # This rule gets checked against the branch object, not the shrubbery
    Attribute('store', lambda user: user.profile.branch.store),
)
```

2.5 Combining Rules Together

All of the rules that we've seen so far are quite simple, each only checking one thing. Fortunately, Bridgekeeper rules can be combined together, letting us model much more complex requirements.

We do this using the `&`, `|` and `~` operators. (If you've used `Q` objects, combining Bridgekeeper rules will feel familiar.)

- Prefixing a rule with `~` inverts it. For example, the expression `~is_apprentice` returns a rule that allows access to everyone that is **not** an apprentice shrubber.
- Combining two rules with `|` allows access if *either* rule matches. For example, `is_staff | is_shrubber` allows access to users that are either administrative staff **or** shrubbers.
- Combining two rules with `&` allows access if *both* rules match. For example, `is_staff & is_shrubber` allows access to users that are both administrative staff **and** shrubbers.

For a more complex example, let's say that we needed to model the following requirement:

> Administrative staff (with `is_staff` set) can edit all shrubberies in the system. Shrubbers can edit all shrubberies in the store they belong to. Apprentice shrubbers can edit all shrubberies in their branch.

First, we need to rephrase this requirement so that it's made up of simpler rules combined with **and**, **or**, and **not**.

> Users can edit shrubberies if:
>
> - They are administrative staff (with `is_staff` set), **or**
> - They are a shrubber, **and** the shrubbery belongs to the same store as them, **or**
> - They are an apprentice shrubber, **and** the shrubbery belongs to the same branch as them

In earlier sections of this chapter, we've already talked about rules that allow access to staff users and users with particular roles. We've also already discussed rules that allow access only to shrubberies belonging to the same store or branch as the user trying to access them. All we need to do now is combine them together:

Listing 2.7: shrubberies/permissions.py

```
from bridgekeeper.rules import is_staff
from .rules import is_shrubber, is_apprentice
from . import models

perms['shrubbery.update_shrubbery'] = is_staff | (
    is_shrubber & Relation(
        'branch',
        Attribute('store', lambda user: user.profile.branch.store),
```

```
    )
) | (
    is_apprentice & Attribute('branch', lambda user: user.profile.branch)
)
```

CHAPTER 3

Using Permissions In Views

Now that we've got our permissions defined, we need to write views that actually use them. If you've already used Django's built-in permission mechanism, Bridgekeeper integrates with that:

Listing 3.1: shrubberies/views.py

```
from django.http import Http404
from django.shortcuts import get_object_or_404
from django.template.response import TemplateResponse

from . import models

def shrubbery_edit(request, shrubbery_id):
    shrubbery = get_object_or_404(models.Shrubbery, id=shrubbery_id)
    if not request.user.has_perm('shrubberies.update_shrubbery', shrubbery):
        raise Http404()
    return TemplateResponse(request, 'shrubbery_edit.html', {
        'shrubbery': shrubbery,
    })
```

We can also check permissions directly through Bridgekeeper. Remember, `bridgekeeper.perms` is more or less just a dict, so we can pull it out of there and call the rule's *`check()`* method:

Listing 3.2: shrubberies/views.py

```
from bridgekeeper import perms

def shrubbery_edit(request, shrubbery_id):
    # ...
    if not perms['shrubberies.update_shrubbery'].check(request.user, shrubbery):
        raise Http404()
    # ...
```

Note: If you use Django's `has_perm()`, like in our first example, Django will consult *all* of your authentication backends to check permissions. For instance, if you've assigned permissions to users in your database through

Django's built-in `user_permissions`, they'll be checked as well. Similarly, if you have a third-party authentication backend (e.g. for social media, LDAP or Active Directory integration) that provides some form of permission checking, that will be checked too.

If you use Bridgekeeper directly, like in our second example, only Bridgekeeper permissions will be checked; in most cases this is what you want.

3.1 Filtering QuerySets

If we're displaying a list, we can also filter a QuerySet so that it only contains objects that the currently-logged-in user holds a certain permission on.

Listing 3.3: shrubberies/views.py

```
from bridgekeeper import perms
from django.core.paginator import Paginator
from django.template.response import TemplateResponse

from . import models

def shrubbery_list(request, shrubbery_id):
    all_shrubberies = models.Shrubbery.objects.all()
    shrubberies = perms['shrubberies.view_shrubbery'].filter(request.user, all_
↪shrubberies)

    # 'shrubberies' is just a regular queryset, so we can do anything
    # we would do with a normal queryset; in this case, let's paginate it
    paginator = Paginator(shrubberies, 10)
    page = paginator.page(1)

    return TemplateResponse(request, 'shrubbery_list.html', {
        'paginator': paginator,
        'page': page,
        'shrubberies': page.object_list,
    })
```

3.2 Class-Based Views

All of the examples we've used so far have been function-based views. Of course, everything that we've covered so far will work inside a class-based view, but Bridgekeeper also comes with a handy shortcut in the form of *QuerySetPermissionMixin*.

Listing 3.4: shrubberies/views.py

```
from bridgekeeper.mixins import QuerySetPermissionMixin
from django.views.generic import ListView, UpdateView

from . import models


class ShrubberyListView(QuerySetPermissionMixin, ListView):
    model = models.Shrubbery
    permission_name = 'shrubberies.view_shrubbery'
```

```
class ShrubberyUpdateView(QuerySetPermissionMixin, UpdateView):
    model = models.Shrubbery
    permission_name = 'shrubberies.update_shrubbery'
```

That's all there is to it; these two views will now only show shrubberies that the currently-logged-in user has permission to view.

3.3 What next?

That's the end of the tutorial; you should now be able to get started modelling your permissions with Bridgekeeper now!

You can read about the other ways you can check permissions, including more convenience shortcuts you can enable and ways to check things like whether somebody *could, hypothetically, have* a permission in the *Checking Permissions* guide. Or, find out more detail about writing rules and permissions in the *Writing Rules and Permissions* guide.

If there's something that you don't understand after following through this tutorial, or that you think could be explained better, please file a documentation bug so that we can improve the docs for future users.

CHAPTER 4

Writing Rules and Permissions

In Bridgekeeper, a **rule** is something that is given a user and a resource, and either **allows** or **blocks** access to the resource. Rules are instances of the *Rule* class (or rather, subclasses of that class), and can be combined together into composite rules.

A Bridgekeeper **permission** consists of a name, usually conforming to Django permission name conventions e.g. `shrubberies.update_shrubbery`, and a rule. Permissions are created by assigning a rule instance to a name in `bridgekeeper.perms`, which acts like a dictionary:

```
from bridgekeeper.rules import Attribute, is_staff
from bridgekeeper import perms

perms['foo.update_widget'] = is_staff
```

The *rules* module provides a range of pre-made rule instances as well as rule classes you can instantiate, as shown above. You can also combine rules using the `&` (and), `|` (or), and `~` (not) operators:

```
perms['foo.view_widget'] = is_staff | Attribute(
    'company', lambda user: user.company)
```

Finally, if none of the built-in rules do what you want, you can subclass *Rule* yourself and write your own.

4.1 Blanket Rules

We introduced what blanket rules are, as well as how to write a custom one, in the *Blanket Rules* section of the tutorial. There, we defined one rule for each role, but if we had more than two roles that might get a bit repetitive.

If you need your blanket rules to take arguments, the easiest way is to write a function that *returns* a rule, like so:

Listing 4.1: shrubberies/rules.py

```
from bridgekeeper.rules import blanket_rule

def has_role(role):
```

```
    def checker(user):
        return user.profile.role == role

    return blanket_rule(checker, repr_string=f"has_role({role!r})")
```

In this case, we're using the optional `repr_string` argument to override how the rule is displayed when debugging, so that we can see what the `role` argument is. (We're using PEP 498 f-strings here, which are supported in Python 3.6+, but you don't have to.)

CHAPTER 5

Checking Permissions

There are two ways to check which permissions a user has using Bridgekeeper.

- Use the methods on the `User` model, which consult Bridgekeeper via its integration into Django's pluggable authorisation system. You can only make the types of checks Django has built-in support for this way, which means you can't check against QuerySets. Also, if you have multiple different authorisation backends (including Django's built in `ModelBackend`), these methods will consult all of them.
- Check against permissions in Bridgekeeper directly. This is the only way to filter QuerySets according to a permission; this method always uses the permissions defined in Bridgekeeper as a single source of truth and does not consult other backends.

5.1 Checking Permissions on an Object

Given an instance of our `Shrubbery` model called `shrubbery`, and a `User` instance `user`, here's how we'd check to see whether the user has permission to update it:

```
from bridgekeeper import perms

# through Django:
user.has_perm('shrubberies.update_shrubbery', obj=shrubbery)
# or through Bridgekeeper:
perms['shrubberies.update_shrubbery'].check(user, shrubbery)
```

Both of these expressions will return either `True` or `False`. Aside from the caveat described above regarding authorisation backends other than Bridgekeeper, these two calls are equivalent; in fact, when you call `has_perm()`, Django will trigger a call to *`check()`* under the hood.

5.2 Checking Permissions on a QuerySet

Of course, Bridgekeeper's headline feature is that it works with QuerySets; given a user and a permission, it can filter down a QuerySet to only return instances for which the user holds the permission.

All we need to do is call *filter()* instead of *check()*, and pass it a QuerySet instead of a single model instance:

```
qs = models.Shrubbery.objects.all()
filtered_qs = perms['shrubberies.view_shrubbery'].filter(user, qs)
```

Bridgekeeper's *filter()* takes any QuerySet, and returns another normal QuerySet (it actually just calls the QuerySet's `filter()` method internally). This means you can call `filter()`, `exclude()` or `order_by()` your QuerySet before you pass it in, or you can `filter()`, `exclude()`, `order_by()`, slice or paginate the QuerySet that Bridgekeeper returns to you.

5.3 Checking Permissions For All Possible Instances

Django's `has_perm()` (and thus also Bridgekeeper's *check()*) allows supplying only a permission name, and not an object instance:

```
user.has_perm('shrubberies.view_shrubbery')
# or,
perms['shrubberies.view_shrubbery'].check(user)
```

Once again, these calls are equivalent, aside from the caveat described above regarding authorisation backends other than Bridgekeeper.

When you check permissions like this without supplying an instance, Bridgekeeper will return `True` if and only if the user has that permission **for every possible instance that could ever exist**. (This is not the same thing as checking whether the user has the permission for every instance *currently in the database*; in fact, this check doesn't actually hit the database at all.)

As an example of this, let's say that the `shrubberies.view_shrubbery` permission was defined to allow staff users access to all shrubberies, and everyone else access to shrubberies in their own branch:

```
perms['shrubberies.view_shrubbery'] = is_staff | Attribute(
    'branch', lambda user: user.profile.branch,
)
```

In this case, the check would return `True` for a staff user, since they will always have access to every possible shrubbery. It will return `False` for a regular user, even if every shrubbery currently in the database belongs to their branch, because it is possible for a shrubbery to be created that belongs to a different branch, which they would then be blocked from editing.

5.4 Checking Permissions For *Any* Possible Instances

Bridgekeeper also provides a second method, `is_possible_for()`, which is the opposite of the above behaviour, in a way:

```
perms['shrubberies.update_shrubbery'].is_possible_for(user)
```

This check will return `True` if and only if the user could possibly have that permission for **any possible instance that could exist**. (Once again, this is not the same as checking whether the user has the permission for at least one instance *currently in the database*, and once again it doesn't actually hit the database at all.)

As an example of this, let's say that the `shrubberies.view_shrubbery` permission was defined to allow only shrubbers to edit shrubberies inside their own branch, using the `is_shrubber` rule we created in the *Blanket Rules* section of the tutorial and combining it with an *Attribute* check:

```
perms['shrubberies.view_shrubbery'] = is_shrubber & Attribute(
    'branch', lambda user: user.profile.branch,
)
```

In this case, the check will return `False` for a user with the `'apprentice'` role, because only users with the `'shrubber'` role can access anything. It will always return `True` for a shrubber, however, even if there are no shrubberies belonging to their branch currently in the database, beacuse it is possible for a shrubbery to exist that belongs to their branch, which they would then be allowed to edit.

Note: The behaviours in this section are effectively implemented by checking whether a permission is always allowed (in the case of *`check()`*) or always denied (in the case of `is_possible_for()`) due to the presence of blanket rules.

In normal use, these methods should always behave how you'd expect. However, if you create a combination of rules that just happens to be tautological for a particular user, Bridgekeeper isn't clever enough to detect that.

This also means that the checks described in this section usually won't need to hit the database.

5.4.1 `has_module_perms()`

Bridgekeeper also supports Django's `has_module_perms()` method. The following call:

```
user.has_module_perms('shrubberies')
```

is equivalent to calling `is_possible_for()` on every permission whose name begins with `shrubberies.`, and returning `True` if any one of them returns `True`.

5.5 Permission Check Summary

Meaning	Django	Bridgekeeper
User has permission `foo.bar` for object `x`	`u.has_perm('foo.bar', x)`	`perms['foo.bar'].check(u, x)`
User has permission `foo.bar` for all possible objects	`u.has_perm('foo.bar')`	`perms['foo.bar'].check(u)`
It is possible for the user to have permission `foo.bar` for some object	*n/a*	`perms['foo.bar'].is_possible_for(u)`
It is possible for the user to have some permission `foo.*` for some object	`u.has_module_perms('foo')`	*n/a*
Filter the queryset `qs` to only the objects that the user has permission `foo.bar` for	*n/a*	`perms['foo.bar'].filter(u, qs)`

5.6 Using permissions in views

Bridgekeeper provides a `QuerySetPermissionMixin`, which will filter a view down to only objects that the currently logged-in user has access to. It works on `ListView`, `DetailView`, and most views that operate on the database except `CreateView`, and is used like this:

```
from bridgekeeper.mixins import QuerySetPermissionMixin

class MyView(QuerySetPermissionMixin, DetailView):
    permission_name = 'applicants.view_applicant'
    model = Applicant
```

Caution: `QuerySetPermissionMixin` will return 404 both for objects that don't exist and objects the user can't access. It might be tempting to try to distinguish between an the two, by returning e.g. 404 for the former and 403 for the latter. Generally, though, it's desirable from a security perspective to not let the user tell the difference between these two cases unless you really need to.

If you're concerned about users getting unexpected 404s when they try to access a page without being logged in, one alternative is to reword your `404.html` accordingly, or even embed a login form there if users aren't logged in.

Bridgekeeper also provides `CreatePermissionGuardMixin`, which will validate unsaved model instances in a `CreateView` (or any subclass of `ModelFormView`) against a given permission, and raise `SuspiciousOperation`, thus preventing the call to `.save()`, if it does not pass. It's used like this:

```
from bridgekeeper.mixins import CreatePermissionGuardMixin

class MyView(CreatePermissionGuardMixin, CreateView):
    permission_name = 'applicants.create_applicant'
    model = Applicant
```

Note: Unlike `QuerySetPermissionMixin`, `CreatePermissionGuardMixin` is only a safety net; you still need to write your forms and views so that a user can't create instances they shouldn't be allowed to, but the mixin will protect you against logic errors in your code, possibly combined with malicious users.

CHAPTER 6

Django REST Framework integration

6.1 Installation

If you want to use Django REST Framework and Bridgekeeper together, you'll need to add the following to your `settings.py`:

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': (
        'bridgekeeper.rest_framework.RulePermissions',
    ),
    'DEFAULT_FILTER_BACKENDS': ('bridgekeeper.rest_framework.RuleFilter',),
}
```

Warning: These settings only set the *default* permission classes and filter backends. If you override either `permission_classes` or `filter_backends` in any `APIView` or `ViewSet` subclass, you'll need to make sure Bridgekeeper's classes are included in those locations too.

6.2 Permission Naming

Once you've changed your settings, all of your API views will automatically apply the appropriate permissions. In order for them to do so, they need to be named according to the conventional Django permission naming scheme. Given a Django app called `app_name` and a model called `ModelName`, the following permissions will be checked:

- `app_name.view_modelname` for all requests.
- `app_name.add_modelname` for `POST` requests.
- `app_name.change_modelname` for `PUT` and `PATCH` requests.
- `app_name.delete_modelname` for `DELETE` requests.

One side-effect of this is that your API consumers will not be able to make changes if they have `add`, `change` or `delete` permissions on some object but don't also have `view` permissions for that same object. That being said, it doesn't make sense for a user to be able to change something they can't see anyway.

CHAPTER 7

Rules

Rule library that forms the core of Bridgekeeper.

This module defines the *Rule* base class, as well as a number of built-in rules.

7.1 The `Rule` API

class bridgekeeper.rules.**Rule**
Base class for rules.

All rules are instances of this class, but not directly; use (or write!) a subclass instead, as this class will raise `NotImplementedError` if you try to actually do anything with it.

check(*user*, *instance=None*)
Check if a user satisfies this rule.

Given a user, return a boolean indicating if that user satisfies this rule for a given instance, or if none is provided, every instance.

filter(*user*, *queryset*)
Filter a queryset to instances that satisfy this rule.

Given a queryset and a user, this method will return a filtered queryset that contains only instances from the original queryset for which the user satisfies this rule.

Parameters

- **queryset** (*django.db.models.QuerySet*) – The initial queryset to filter
- **user** (*django.contrib.auth.models.User*) – The user to match against.

Returns A filtered queryset

Return type django.db.models.QuerySet

> **Warning:** If you are subclassing this class, don't override this method; override *query()* instead.

is_possible_for(*user*)
Check if it is possible for a user to satisfy this rule.

Returns `True` if it is possible for an instance to exist for which the given user satisfies this rule, `False` otherwise.

For example, in a multi-tenanted app, you might have a rule that allows access to model instances if a user is a staff user, or if the instance's tenant matches the user's tenant.

In that case, *check()*, when called without an instance, would return `True` only for staff users (since only they can see *every* instance). This method would return `True` for all users, because every user could possibly see an instance (whether it's one that exists currently in the database, or a hypothetical one that might in the future).

Cases where this method would return `False` include where a user doesn't have the right role or subscription plan to use a feature at all; this method is the single-permission equivalent of has-module-perms.

7.2 Built-in Blanket Rules

bridgekeeper.rules.**always_allow**
Rule that always allows access to everything.

bridgekeeper.rules.**always_deny**
Rule that never allows access to anything.

bridgekeeper.rules.**is_authenticated**
Rule that allows access to users for whom `is_authenticated` is `True`.

bridgekeeper.rules.**is_superuser**
Rule that allows access to users for whom `is_superuser` is `True`.

bridgekeeper.rules.**is_staff**
Rule that allows access to users for whom `is_staff` is `True`.

bridgekeeper.rules.**is_active**
Rule that allows access to users for whom `is_active` is `True`.

7.3 Rule Classes

class bridgekeeper.rules.**Attribute**(*attr*, *matches*)
Rule class that checks the value of an instance attribute.

This rule is satisfied by model instances where the attribute given in `attr` matches the value given in `matches`.

Parameters

- **attr** (*str*) – An attribute name to match against on the model instance.
- **value** – The value to match against, or a callable that takes a user and returns a value to match against.

For instance, if you had a model class `Widget` with an attribute `colour` that was either `'red'`, `'green'` or `'blue'`, you could limit access to blue widgets with the following:

```
blue_widgets_only = Attribute('colour', matches='blue')
```

Restricting access in a multi-tenanted application by matching a model's `tenant` to the user's might look like this:

```
applications_by_tenant = Attribute('tenant',
                                   lambda user: user.tenant)
```

Warning: This rule uses Python equality (==) when checking a retrieved Python object, but performs an equality check on the database when filtering a QuerySet. Avoid using it with imprecise types (e.g. floats), and ensure that you are using the correct Python type (e.g. `decimal.Decimal` for decimals rather than floats or strings), to prevent inconsistencies.

class bridgekeeper.rules.**Relation**(*attr*, *rule*)

Check that a rule applies to a ForeignKey.

Parameters

- **attr** (*str*) – Name of a foreign key attribute to check.
- **rule** (Rule) – Rule to check the foreign key against.

For example, given `Applicant` and `Application` models, to allow access to all applications to anyone who has permission to access the related applicant:

```
perms['foo.view_application'] = Relation(
    'applicant', perms['foo.view_applicant'])
```

class bridgekeeper.rules.**ManyRelation**(*query_attr*, *rule*)

Check that a rule applies to a many-object relationship.

This can be used in a similar fashion to *Relation*, but across a `ManyToManyField`, or the remote end of a `ForeignKey`.

Parameters

- **query_attr** (*str*) – Name of a many-object relationship to check. This This is the name that you use when filtering this relationship using `.filter()`. If you are on the side of the relationship where the field is defined, this is typically the lowercased model name (e.g. `mymodel` on its own, *not* `mymodel_set`), unless you've set `related_name` or `related_query_name`.
- **rule** (Rule) – Rule to check the foreign object against.

For example, given `Agency` and `Customer` models, to allow agency users access only to customers that have a relationship with their agency:

```
perms['foo.view_customer'] = ManyRelation(
    'agency', Is(lambda user: user.agency))
```

class bridgekeeper.rules.**Is**(*instance*)

Rule class that checks the identity of the instance.

This rule is satisfied only by a the provided model instance.

Parameters **instance** – The instance to match against, or a callable that takes a user and returns a value to match against.

For instance, if you only wanted a user to be able to update their own profile:

```
own_profile = Is(lambda user: user.profile)
```

class bridgekeeper.rules.**In**(*collection*)
Rule class that checks the instance is a member of a collection.

This rule is satisfied only by model instances that are members of the provided collection.

Parameters **collection** – The collection to match against, or a callable that takes a user and returns a value to match against.

For instance, if you only wanted to match groups a user is in:

```
own_profile = Is(lambda user: user.profile)
```

7.4 Built-in rule instances

bridgekeeper.rules.**current_user = Is(<function <lambda>>)**
Rule class that checks the identity of the instance.

This rule is satisfied only by a the provided model instance.

Parameters **instance** – The instance to match against, or a callable that takes a user and returns a value to match against.

For instance, if you only wanted a user to be able to update their own profile:

```
own_profile = Is(lambda user: user.profile)
```

bridgekeeper.rules.**in_current_groups = In(<function <lambda>>)**
Rule class that checks the instance is a member of a collection.

This rule is satisfied only by model instances that are members of the provided collection.

Parameters **collection** – The collection to match against, or a callable that takes a user and returns a value to match against.

For instance, if you only wanted to match groups a user is in:

```
own_profile = Is(lambda user: user.profile)
```

7.5 Extension Points (For Writing Your Own `Rule` Subclasses)

class bridgekeeper.rules.**Rule**
If you want to create your own rule class, these are the methods you need to override.

query(*user*)
Generate a Q object.

Note: This method is used internally by *filter()*; subclasses will need to override it but you should never need to call it directly.

Given a user, return a Q object which will filter a queryset down to only instances for which the given user satisfies this rule.

Alternatively, return *UNIVERSAL* if this user satisfies this rule for every possible object, or *EMPTY* if this user cannot satisfy this rule for any possible object. (These two values are usually only returned in "blanket rules" which depend only on some property of the user, e.g. the built-in *is_staff*, but these are usually best created with the `blanket_rule` decorator.)

Parameters **user** (*django.contrib.auth.models.User*) – The user to match against.

Returns A query that will filter a queryset to match this rule.

Return type django.db.models.Q

check(*user*, *instance=None*)

Check if a user satisfies this rule.

Given a user, return a boolean indicating if that user satisfies this rule for a given instance, or if none is provided, every instance.

bridgekeeper.rules.**UNIVERSAL = UNIVERSAL**

bridgekeeper.rules.**EMPTY = EMPTY**

CHAPTER 8

Convenience Helpers

8.1 QuerySet and Manager Classes

class bridgekeeper.querysets.**PermissionQuerySet**(*model=None*, *query=None*, *using=None*, *hints=None*)

A QuerySet subclass that provides a convenience method.

visible_to(*user*, *permission*)

Filter the QuerySet to objects a user has a permission for.

Parameters

- **user** (*django.contrib.auth.models.User*) – User to check permission against.
- **permission** (*str*) – Permission to check.

This method only works with permissions that are defined in `perms`; regular Django row-level permission checkers can't be invoked on the QuerySet level.

It is a convenience wrapper around *filter()*.

8.2 View Mixins

class bridgekeeper.mixins.**CreatePermissionGuardMixin**(*permission_map=None*, **args*, ***kwargs*)

A view that checks permissions before creating model instances.

Use this mixin with `CreateView`, and supply the *permission_name* of a Bridgekeeper permission. Your view will then do two things:

- Check that it's possible for a user to create any new instances at all (i.e. that *is_possible_for()* returns `True` on the supplied permission). If not, the mixin raises `PermissionDenied`.

- Just before the form is saved, checks the unsaved model instance against the supplied permission; if it fails, the mixin raises `SuspiciousOperation`.

Note that unlike *`QuerySetPermissionMixin`*, this mixin won't automatically apply permissions for you. Ideally, your view (or the form class your view uses) should make it impossible for users to create instances they're not allowed to create; fields that must be set to a certain value should be set automatically and not displayed in the form, choice fields should have their `choices` limited to only values the user is allowed to set, and so on.

Bridgekeeper can't (and arguably shouldn't) reach into your form and modify it for you. Instead, this mixin provides a last line of defence; if your view has a bug where a user can create something they're not allowed to, the mixin will prevent the object from actually being created, and crash loudly in a way that your error reporting systems can pick up, allowing you to fix the bug.

permission_name
The name of the Bridgekeeper permission to check against, e.g. `'shrubberies.update_shrubbery'`.

class bridgekeeper.mixins.**QuerySetPermissionMixin**(*permission_map=None*, **args*, ***kwargs*)

View mixin that filters QuerySets according to a permission.

Use this mixin with any class-based view that expects a `get_queryset` method (e.g. `ListView`, `DetailView`, `UpdateView`, or any other views that subclass from `MultipleObjectMixin` or `SingleObjectMixin`), and supply a *`permission_name`* attribute with the name of a Bridgekeeper permission.

The view's queryset will then be automatically filtered to objects that the user requesting the page has the supplied permission for. For multiple-object views like `ListView`, objects the user doesn't have the permission for just won't be in the list. For single-object views like `UpdateView`, attempts to access objects the user doesn't have the permission for will just 404.

permission_name
The name of the Bridgekeeper permission to check against, e.g. `'shrubberies.update_shrubbery'`.

CHAPTER 9

Django REST Framework integration

class bridgekeeper.rest_framework.**BridgekeeperRESTMixin**

Mixin for Django REST Framework integration classes.

get_action(*request*, *view*, *obj=None*)

Return the action that a particular request is performing.

Usually, this is one of `'view'`, `'add'`, `'change'` or `'delete'`. This is used by *get_permission_name()* to generate the name of the appropriate permission.

Returns Name of an action.

Return type str

get_operand_name(*request*, *view*, *obj=None*)

Return the name of the thing that a request is acting on.

The default implementation works if `obj` is a model instance (when it is provided), or if `view` is a view that has either a `queryset` attribute or `get_queryset()` method (otherwise).

This is used by *get_permission_name()* to generate the name of the appropriate permission.

Returns A tuple in the form (app_label, operand_name).

Return type (str, str)

get_permission(*request*, *view*, *obj=None*)

Return a rule object to check against for this request.

The default implementation just looks up the name returned by *get_permission_name()*.

Returns Rule object.

Return type *bridgekeeper.rules.Rule*

get_permission_name(*request*, *view*, *obj=None*)

Return the name of the permission to use for a request.

The default implementation returns a name of the form `'{app_label}.{action}_{operand_name}'`, which will result in something like `'shrubberies.view_shrubber'` or `'shrubberies.delete_shrubbery'`.

`app_label` and `operand_name` are provided by *`get_operand_name()`*, and `action` is provided by *`get_action()`*, so if you need to override this behaviour, it may be easier to override those methods instead.

Returns Permission name.

Return type str

skip_permission_checks(*request*, *view*, *obj=None*)

Skips all permission checks for certain requests.

The default implementation will skip permission checks for the `APIRootView` view class used by the built-in `DefaultRouter`.

Returns Whether to skip permission checks for the given request.

Return type bool

class bridgekeeper.rest_framework.**RuleFilter**

Django REST Framework filter class for Bridgekeeper.

This filter class doesn't expect any client interaction or present any UI to the API explorer; it's simply a mechanism for automatically filtering QuerySets according to Bridgekeeper permissions.

Note that this filter will always check the `view` permission; this means that if a particular user has permissions to edit but not view something, they'll get 404s on everything. That said, it doesn't make much sense for users to have edit but not view permissions on something anyway.

class bridgekeeper.rest_framework.**RulePermissions**

Django REST Framework permission class for Bridgekeeper.

Note that this class **does not**, by itself, perform queryset filtering on list views, since Django REST Framework doesn't provide an API for permission classes to do so.

CHAPTER 10

Changelog

10.1 dev

- **Breaking change:** *Relation* and *ManyRelation* no longer require the model class on the other side of the relation to be passed in as an argument.
- **Breaking change:** *ManyRelation* removes the `attr` argument, requiring only `query_attr`.
- **Breaking change:** Python 3.4 and Django 1.11 are no longer supported.

10.2 0.7

Add *In* permission class, and two predefined rule instances, *current_user* and *in_current_groups*.

10.3 0.5

- Minor Django REST Framework-related fixes.

10.4 0.4

- Added initial support for Django REST Framework.
- Documentation improvements.

10.5 0.3

- Renamed **predicates** to **rules**, because the latter is a more accessible term that describe the concept just as well. Besides, "permissions are made up of rules" sounds a lot better than "permissions are made up of predicates".
- Renamed **ambient predicates** to **blanket rules**, because it's a more descriptive name. Note that the `@ambient` decorator is now called `@blanket_rule`, because having a `@blanket` decorator would be weird.

10.6 0.2

- Renamed `bridgekeeper.registry.registry` to `bridgekeeper.perms`.
- Renamed `bridgekeeper.predicates.Predicate.apply()` to `check()`
- Changed `bridgekeeper.predicates.Predicate.filter()` so that it takes the user object as the first argument, for consistency with the rest of the library (i.e. it's singnature went from `filter(queryset, user)` to `filter(user, queryset)`).

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index

U

V