

---

# **django-authority Documentation**

***Release 0.14***

**The django-authority team**

**Feb 07, 2020**



---

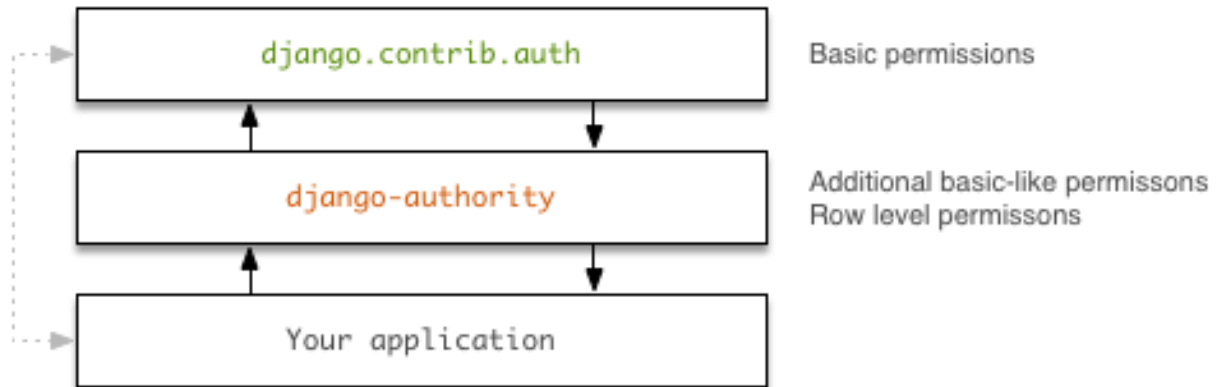
## Contents

---

<b>1</b>	<b>Documentation</b>	<b>3</b>
<b>2</b>	<b>Other pages</b>	<b>17</b>
	<b>Index</b>	<b>21</b>



django-authority is a powerful layer between Django's basic permission system (provided through `django.contrib.auth`) and your application:



This application provides three abilities:

1. It gives you the ability to add permissions like Django's generic permissions to any kind of model without having to add them to the model's Meta class.
2. It provides a very simple way to create per-object-permissions. You might be more familiar with the term *row level permissions*.
3. It wraps Django's generic permissions so you can use the same syntax as for the options above. But note that django-authority does not add any voodoo-code to Django's `contrib.auth` system, it keeps your existing permission system intact!

django-authority uses a cache that is stored on the user object to help improve performance. However, if the `Permission` table changes the cache will need to be invalidated. More information about this can be found in the tips and tricks section.

**Warning:** We have just started with the documentation and it's far from being perfect. If you find glitches, errors or just have feedback, please contact the team: [Support](#).



---

**Note:** The create-permission topics are based on each other. If you are new to django-authority we encourage to read from top to bottom.

---

### Installation topics:

## 1.1 Installation

The installation of django-authority is easy. Whether you want to use the latest stable or development version, you have the following options.

### 1.1.1 The latest stable version

The latest, stable version is always available via the *Python package index* (PyPI). You can download the latest version on [the site](#) but most users would prefer either `pip` or `easy_install`:

```
pip install django-authority

# .. or with easy_install:

easy_install django-authority
```

### 1.1.2 Development version

The latest development version is located on it's [Github account](#). You can checkout the package using the [Git](#) scm:

```
git clone https://github.com/jazzband/django-authority
```

Then install it manually:

```
cd django-authority
python setup.py install
```

**Warning:** The development version is not fully tested and may contain bugs, so we prefer to use the latest package from pypi.

## 1.2 Configuration

### 1.2.1 settings.py

To enable django-authority you just need to add the package to your `INSTALLED_APPS` setting within your `settings.py`:

```
# settings.py
INSTALLED_APPS = (
    ...
    'authority',
)
```

Make sure your `settings.py` contains the following settings to enable the context processors:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    'django.core.context_processors.auth',
    'django.core.context_processors.debug',
    'django.core.context_processors.i18n',
    'django.core.context_processors.media',
    'django.core.context_processors.request',
)
```

django-authority defaults to using a smart cache when checking permissions. This can be disabled by adding the following line to `settings.py`:

```
AUTHORITY_USE_SMART_CACHE = False
```

### 1.2.2 urls.py

You also have to modify your root `URLConf` (e.g. `urls.py`) to include the app's URL configuration and automatically discover all the permission classes you defined:

```
from django.contrib import admin
import authority

admin.autodiscover()
authority.autodiscover()

# ...

urlpatterns += patterns('',
    (r'^authority/', include('authority.urls')),
)
```



If you're using Django 1.1 this will automatically add a [site-wide action](#) to the admin site which can be removed as shown here: [Handling permissions using Django's admin interface](#).

That's all (for now).

**Create and check permissions:**

## 1.3 Create a basic permission

### 1.3.1 Where to store permissions?

First of all: All following permission classes should be placed in a file called `permissions.py` in your application. For the *why* please have a look on [How permissions are discovered](#).

### 1.3.2 Basic permissions

Let's start with an example:

```
import authority
from authority import permissions
from django.contrib.flatpages.models import FlatPage

class FlatpagePermission(permissions.BasePermission):
    label = 'flatpage_permission'

authority.register(FlatPage, FlatpagePermission)
```

Let's have a look at the code above. First of, if you want to create a new permission you have to subclass it from the `BasePermission` class:

```
from authority import permissions
class FlatpagePermission(permissions.BasePermission):
    # ...
```

Next, you need to name this permission using the `label` attribute:

```
class FlatpagePermission(permissions.BasePermission):
    label = 'flatpage_permission'
```

And finally you need to register the permission with the pool of all other permissions:

```
authority.register(FlatPage, FlatpagePermission)
```

The syntax of this is simple:

```
authority.register(<model>, <permission_class>)
```

While this is not much code, you already wrapped Django's basic permissions (`add_flatpage`, `change_flatpage`, `delete_flatpage`) for the model `FlatPage` and you are ready to use it within your templates or code:

---

**Note:** See [Django's basic permissions](#) how Django creates this permissions for you.

---

### 1.3.3 Example permission checks

This section shows you how to check for Django's basic permissions with django-authority.

#### In your python code

```
def my_view(request):
    check = FlatPagePermission(request.user)
    if check.change_flatpage():
        print "Yay, you can change a flatpage!"
```

#### Using the view decorator

```
from authority.decorators import permission_required_or_403

@permission_required_or_403('flatpage_permission.change_flatpage')
def my_view(request):
    # ...
```

See *Check permissions using the decorator* how the decorator works in detail.

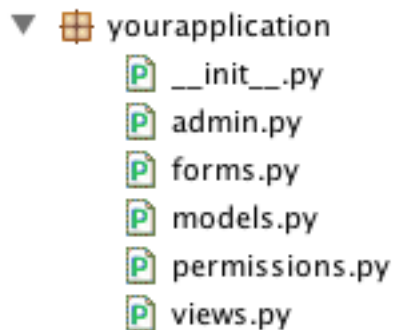
#### In your templates

```
{% ifhasperm "flatpage_permission.change_flatpage" request.user %}
    Yay, you can change a flatpage!
{% else %}
    Nope, sorry. You aren't allowed to change a flatpage.
{% endifhasperm %}
```

See *Check permissions in templates* how the templatetag works in detail.

### 1.3.4 How permissions are discovered

On first runtime of your Django project `authority.autodiscover()` will load all `permissions.py` files that are in your `settings.INSTALLED_APPS` applications. See *Configuration* how to set up `autodiscover`.



We encourage you to place your permission classes in a file called `permissions.py` inside your application directories. This will not only keep your application files clean, but it will also load every permission class at runtime when used with `authority.autodiscover()`.

If you really want, you can place these permission-classes in other files that are loaded at runtime. `__init__.py` or `models.py` are such files.

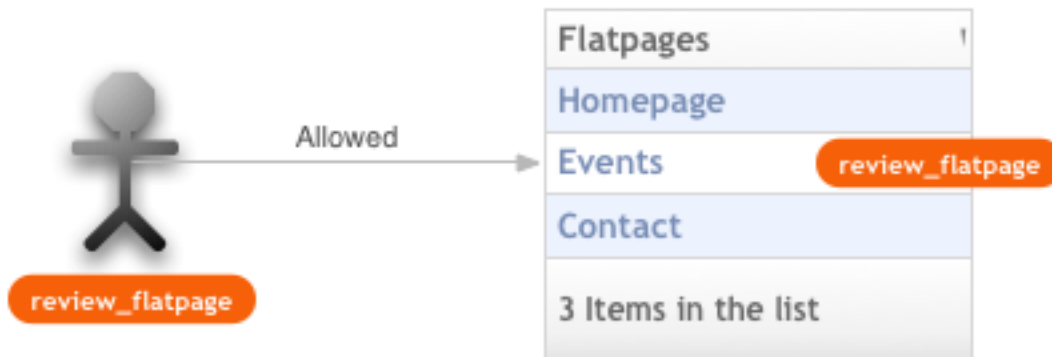
## 1.4 Create a per-object permission

django-authority provides a super simple but nifty feature called *per-object permission*. A description would be:

```
Attach a <codename> to an object
Attach a <codename> to an user
```

```
If the user has <codename> and the object has <codename> then do-something,
otherwise do-something-else.
```

This might sound strange but let's have a closer look on this pattern. In terms of users and flatpages a visual example would be:



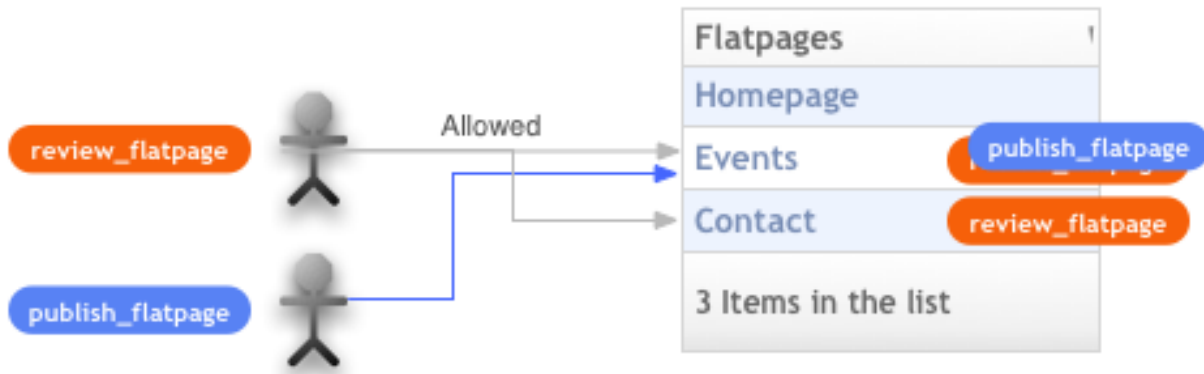
The user is allowed to review the flatpage “Events”.

You are not limited to a 1:1 relation, you can add this codename to multiple objects:



The user is allowed to review the flatpages “Events” and “Contact”.

And you can do this with any objects in any direction:



The user is allowed to review the flatpages “Events” and “Contact”. Another user is allowed to publish the flatpage “Events”.

### 1.4.1 Create per-object permissions

Creating per-object permissions is super simple. See this piece of permission class code:

```
class FlatPagePermission(BasePermission):
    label = 'flatpage_permission'
    checks = ('review',)

authority.register(FlatPage, FlatPagePermission)
```

This permission class is similar to the one we already created in [Create a basic permission](#) but we added the line:

```
checks = ('review',)
```

This tells the permission class that it has a permission check (or codename) review. Under the hood this check gets translated to `review_flatpage(review_<modelname>)`.

---

**Important:** Be sure that you have understand that we have not written any line of code yet. We just added the codename to the checks attribute.

---

### 1.4.2 Attach per-object permissions to objects

Please see [Handling permissions using Django’s admin interface](#) for this.

### 1.4.3 Check per-object permissions

As we noted above, we have not written any permission comparing code yet. This is your work. In theory the permission lookup for per-object permissions is:

```
if <theuser> has <codename> and <object> has <codename>:
    return True
else:
    return False
```

---

**Important:** The syntax is similar to the permission checks we've already seen in [Create a basic permission](#) for the basic permissions but now we have to pass each function a model instance we want to check!

---

### In your python code

```
from myapp.permissions import FlatPagePermission
def my_view(request):
    check = FlatPagePermission(request.user)
    flatpage_object = Flatpage.objects.get(url='/homepage/')
    if check.review_flatpage(flatpage_object):
        print "Yay, you can change *this* flatpage!"
```

### Using the view decorator

```
from django.contrib.auth import Flatpage
from authority.decorators import permission_required_or_403

@permission_required_or_403('flatpage_permission.review_flatpage',
                           (Flatpage, 'url__iexact', 'url')) # The flatpage_object
def my_view(request, url):
    # ...
```

See [Check permissions using the decorator](#) how the decorator works in detail.

### In your templates

```
{% ifhasperm "flatpage_permission.review_flatpage" request.user flatpage_object %}
    Yay, you can change *this* flatpage!
{% else %}
    Nope, sorry. You aren't allowed to change *this* flatpage.
{% endifhasperm %}
```

See [Check permissions in templates](#) how the template tag works in detail.

## 1.5 Create a custom permission

django-authority allows you to define powerful custom permission. Let's start again with an example code:

```
import authority
from authority import permissions
from django.contrib.flatpages.models import Flatpage

class FlatpagePermission(permissions.BasePermission):
    label = 'flatpage_permission'

authority.register(Flatpage, FlatpagePermission)
```

A custom permission is a simple method of the permission class:

```
import authority
from authority import permissions
from django.contrib.flatpages.models import Flatpage

class FlatpagePermission(permissions.BasePermission):
    label = 'flatpage_permission'
    checks = ('my_custom_check',)

    def my_custom_check(self, flatpage):
        if(flatpage.url == '/about/'):
            return True
        return False

authority.register(Flatpage, FlatpagePermission)
```

Note that we first added the name of your custom permission to the `checks` attribute, like in [Create a per-object permission](#):

```
checks = ('my_custom_check',)
```

The permission itself is a simple function that accepts an arbitrary number of arguments. A permission class should always return a boolean whether the permission is True or False:

```
def my_custom_check(self, flatpage):
    if flatpage.url == '/about/':
        return True
    return False
```

**Warning:** Although it's possible to return other values than `True`, for example an object which also evaluates to `True`, we highly advise to only return booleans.

Custom permissions are not necessary related to a model, you can define simpler permissions too. For example, return `True` if it's between 10 and 12 o'clock:

```
def datetime_check(self):
    hour = int(datetime.datetime.now().strftime("%H"))
    if hour >= 10 and hour <= 12:
        return True
    return False
```

But most often you want to combine such permissions checks. The next example would allow an user to have permission to edit a flatpage only between 8 and 12 o'clock in the morning:

```
def morning_flatpage_check(self, flatpage):
    hour = int(datetime.datetime.now().strftime("%H"))
    if hour >= 8 and hour <= 12 and flatpage.url == '/about/':
        return True
    return False
```

## 1.5.1 Check custom permissions

The permission check is similar to [Create a basic permission](#) and [Create a per-object permission](#).

**Warning:** Although *per-object* permissions are translated to `<permname>_<modelname>` this is not the case for custom permissions! A custom permission `my_custom_check` remains `my_custom_check`.

### In your python code

```
from myapp.permissions import FlatPagePermission
def my_view(request):
    check = FlatPagePermission(request.user)
    flatpage_object = Flatpage.objects.get(url='/homepage/')
    if check.my_custom_check(flatpage=flatpage_object):
        print "Yay, you can change *this* flatpage!"
```

### Using the view decorator

```
from django.contrib.auth import Flatpage
from authority.decorators import permission_required_or_403

@permission_required_or_403('flatpage_permission.my_custom_check',
                           (Flatpage, 'url__iexact', 'url')) # The flatpage_object
def my_view(request, url):
    # ...
```

See *Check permissions using the decorator* how the decorator works in detail.

### In your templates

```
{% ifhasperm "flatpage_permission.my_custom_check" request.user flatpage_object %}
    Yay, you can change *this* flatpage!
{% else %}
    Nope, sorry. You aren't allowed to change *this* flatpage.
{% endifhasperm %}
```

See *Check permissions in templates* how the templatetag works in detail.

### Permission checks in detail

## 1.6 Check permissions in python code

*to be written*

## 1.7 Check permissions using the decorator

**Note:** A decorator is not the ultimate painkiller, if you need to deal with complex permission handling, take a look at *Check permissions in python code*.

### 1.7.1 The decorator syntax

Lets start with an example permission:

```
class FlatpagePermission(permissions.BasePermission):
    label = 'flatpage_permission'
    checks = ('can_do_foo',)

    def can_do_foo(self):
        # ...

authority.register(Campaign, FlatpagePermission)
```

A decorator for such a simple view would look like:

```
from authority.decorators import permission_required

@permission_required('flatpage_permission.can_do_foo')
def my_view(request):
    # ...
```

The decorator automatically takes the user object from the view's arguments and calls `can_do_foo`. If this function returns `True`, the view gets called, otherwise the user will be redirected to the login page.

#### Passing arguments to the permission

You can pass any arguments to the permission function. Assumed our permission function looks like this:

```
def can_do_foo(self, view_arg1, view_arg2=None):
    # ...
```

Our decorator can *grab* the arguments from the view and passes it to the permission function. Just take the arguments from the view and place them as a string on the decorator:

```
@permission_required('flatpage_permission.can_do_foo', 'arg1', 'arg2')
def my_view(required, arg1, arg2):
    # ...
```

What happens under the hood?:

```
# Assumed the view gets called like this
my_view(request, 'bla', 'blubb')

# At the end, the decorator would been called like this
can_do_foo('bla', 'blubb')
```

#### Passing queryset lookups to the permission

You can pass queryset lookups instead of an argument. This might look a bit strange first, but it can save you a ton of code. Instead of passing a simple string to the permission function, declare a tuple of the syntax:

```
(<model>, '<field_lookup>', 'view_arg')
# .. or ..
('<appname>.<modelname>', '<field_lookup>', 'view_arg')
```



Here is an example:

```
# permission.py
def can_do_foo(self, flatpage_instance=None):
    # ...

# views.py
from django.contrib.flatpages.models import Flatpage
@permission_required('flatpage_permission.can_do_foo', (Flatpage, 'url__iexact', 'url
→'))
def flatpage(required, url):
    # ...
```

What happens under the hood? It's nearly the same as the *simple* decorator would do, except that the argument is fetched with a `get_object_or_404` statement. So this is the same:

```
(Flatpage, 'url__iexact', 'url')
get_object_or_404(Flatpage, 'url__iexact='/about/')
```

---

**Note:** For all available field lookups, please refer to the Django documentation: [Field lookups](#)

---

## 1.7.2 Contributed decorators

django-authority contributes two decorators, the syntax of both is the same as described above:

- `permission_required`
- `permission_required_or_403`

In a nutshell, `permission_required_or_403` does the same as `permission_required` except it returns a `Http403` Response instead of redirecting to the login page.

Just like Django's `500.html` and `404.html` you are able to override the template used in the permission denied page. Simply create a `403.html` template in your template directory. It will get the path of the denied page passed as the context variable `request_path`.

## 1.8 Check permissions in templates

django-authority provides a couple of template tags which allows you to get permissions for a user (and a related object).

### 1.8.1 ifhasperm

This function checks whether a permission is True or False for a user and (optional) a related object.

Syntax:

```
{% ifhasperm [permission_label].[check_name] [user] [*objs] %}
    lalala
{% else %}
    meh
{% endifhasperm %}
```

Example:

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

## 1.8.2 get\_permissions

Retrieves all permissions associated with the given obj and user and assigns the result to a context variable.

Syntax and example:

```
{% get_permissions obj %}
{% for perm in permissions %}
    {{ perm }}
{% endfor %}

{% get_permissions obj as "my_permissions" %}
{% get_permissions obj for request.user as "my_permissions" %}
```

## 1.8.3 get\_permission

Performs a permission check with the given signature, user and objects and assigns the result to a context variable.

Syntax:

```
{% get_permission [permission_label].[check_name] for [user] and [objs] as [varname]
↪ %}
```

Example:

```
{% get_permission "poll_permission.change_poll" for request.user and poll as "is_
↪ allowed" %}
{% get_permission "poll_permission.change_poll" for request.user and poll,second_poll_
↪ as "is_allowed" %}

{% if is_allowed %}
    I've got ze power to change ze polllllllzzz. Muahahaa.
{% else %}
    Meh. No power for meeeee.
{% endif %}
```

**Permission assigning and handling**

## 1.9 Handling permissions in python code

*to be written*

## 1.10 Handling permissions using Django’s admin interface

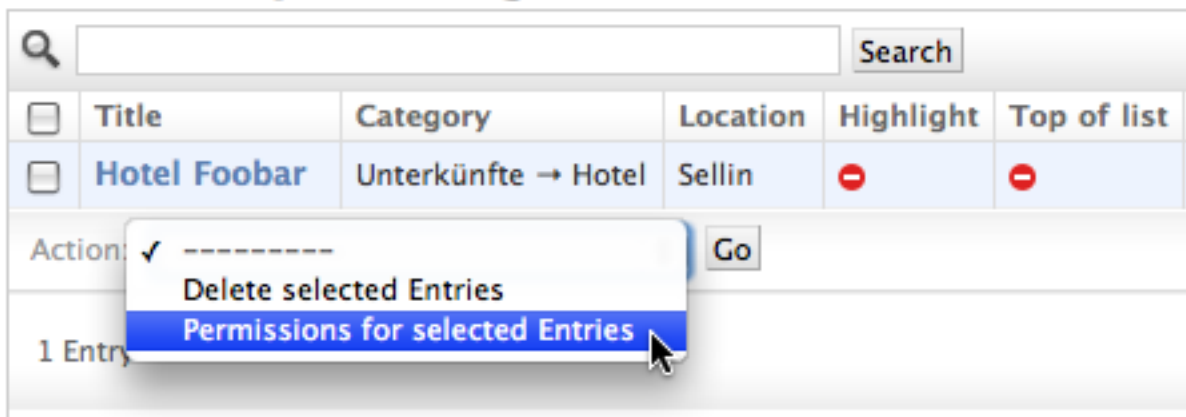
*to be written*

**Note:** Django admin actions are available in Django 1.1 or later.

### 1.10.1 Apply permissions using Django’s admin actions

This feature is limited to superusers and users with either the “Can change permission” (`change_permission`) or the “Can change foreign permission” (`change_foreign_permission`) [permission](#).

#### Select Entry to change



#### Disable the admin action site-wide

To disable the action site-wide, place this line somewhere in your code. One of your app `admin.py` files might be a good place:

```
admin.site.disable_action('edit_permissions')
```

Further informations are available in Django’s documentation: [Disabling a site-wide action](#).

#### Disable the admin action per ModelAdmin instance

In case you want to disable the permission action per `ModelAdmin`, delete this action within the `get_actions` method. Here is an example:

```
class EntryAdmin(admin.ModelAdmin):
    def get_actions(self, request):
        actions = super(EntryAdmin, self).get_actions(request)
        del actions['edit_permissions']
        return actions
```

Further informations are available in Django’s documentation: [Conditionally enabling or disabling actions](#).

## 1.11 Handling permissions using templates

*to be written*

- search
- genindex

## 2.1 Hints, tips and tricks

Within a permission class, you can refer to the user and group using `self`:

```
class CampaignPermission(permissions.BasePermission):
    label = 'campaign_permission'
    checks = ('do_foo',)

    def do_foo(self, campaign=None):
        print self.user
        print self.group
        # ...
```

You can unregister permission classes and re-register them:

```
authority.unregister(Campaign)
authority.register(Campaign, CampaignPermission)
```

Within a permission class, you can refer to Django's basic permissions:

```
class FlagpagePermisson(permissions.BasePermission):
    label = 'flatpage_permission'
    checks = ('do_foo',)

    def do_foo(self, campaign=None):
        if foo and self.change_flatpage():
            # ...
```

```
authority.register(Flatpage, FlagpagePermisson)
```

If the `Permission` table changes during the lifespan of a django-authority permission instance and the smart cache is being used, you will need to call `invalidate_permissions_cache` in order to see that changes:

```
class UserPermission(permission.BasePermission):
    label = 'user_permission'
    checks = ('do_foo',)
authority.register(User, UserPermission)

user_permission = UserPermission(user)

# can_foo is False here since the permission has not yet been added.
can_foo = user_permission.has_user_perms('foo', user)

Permission.objects.create(
    content_type=Permission.objects.get_content_type(User),
    object_id=user.pk,
    codename='foo',
    user=user,
    approved=True,
)

# can_foo is still False because the permission cache has not been
invalidated yet.
can_foo = user_permission.has_user_perms('foo', user)

user_permission.invalidate_permissions_cache()

# can_foo is now True
can_foo = user_permission.has_user_perms('foo', user)
```

This is particularly useful if you are using the permission instances during a request, where it is unlikely that the state of the `Permission` table will change.

Although the previous example was only passing in a `user` into the permission, smart caching is used when getting permissions in a `group` as well.

## 2.2 Support

We've created a [google group](#) for django-authority. If you have questions or suggestions, please drop us a note.

For more specific issues and bug reports please use the [issue tracker](#) on django-authority's Github page.

**Warning:** This document is for internal use only.

## 2.3 Documentation Guildlines

### 2.3.1 Headline scheme

```
=====
First level (equals top and bottom)
=====
```

(continues on next page)

(continued from previous page)

```
Second Level (equals bottom)
=====

Third level (dashes botton)
-----

Fourth level (drunken dashes bottom)
~~~~~
```

Please try to use not more than 4 levels of headlines.

## 2.3.2 Overall salutation guidelines

Use the *We* and *you*:

```
We think that you should send us a bottle of your local beer.
```

## 2.3.3 Some thoughts

- Many internal links are good
- Text should not be wider than 80 characters
- Two pages are better than one ultra-long page





### A

`autodiscover`, [4](#), [5](#)

### B

`BasePermission`, [5](#)

### G

`get_permission`, [13](#)

`get_permissions`, [13](#)

### I

`ifhasperm`, [13](#)

### P

`permission_required`, [11](#)

`permission_required_or_403`, [11](#)

`permissions.py`, [5](#)

### S

`settings.py`, [4](#)

`Support`, [18](#)

### U

`urls.py`, [4](#)