django-permissions-auditor Documentation

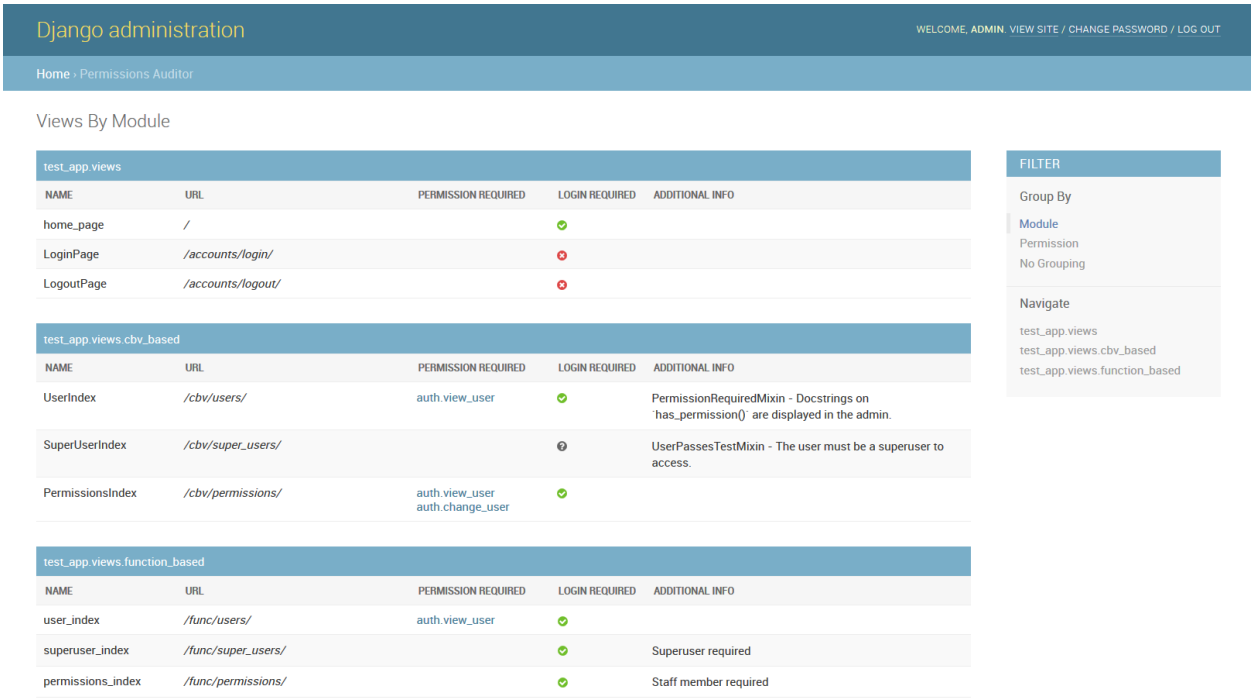
Release 0.3.3

Christian Klus

Jan 09, 2019

Contents:

Admin site for auditing and managing permissions for views in your Django app.



CHAPTER 1

Features

- Automatically parse views registered in Django's URL system
- Out of the box support for Django's authentication system
- Easily extensible for custom permission schemes

CHAPTER 2

Installation

Requirements:

- Django >=2.1
- Python 3.5, 3.6, 3.7

To install:

```
pip install django-permissions-auditor
```

Add *permissions_auditor* to your `INSTALLED_APPS` in your project's `settings.py` file:

```
INSTALLED_APPS = [
    ...
    'permissions_auditor',
    ...
]
```

2.1 Contribute

- Issue Tracker: https://github.com/AACEngineering/django-permissions-auditor/issues
- Source Code: https://github.com/AACEngineering/django-permissions-auditor

2.2 License

The project is licensed under the MIT license.

2.2.1 Overview

In large Django applications that require complex access control, it can be difficult for site administrators to effectively assign and manage permissions for users and groups.

I often found that I needed to reference my site's source code in order to remember what permission was required for what view - something end-users and managers shouldn't need to do.

Django-permissions-auditor attempts to solve this problem by automatically parsing out permissions so that administrators can easily manage their site.

2.2.2 Installation

Requirements:

- Django >=2.1
- Python 3.5, 3.6, 3.7

To install:

```
pip install django-permissions-auditor
```

Add *permissions_auditor* to your `INSTALLED_APPS` in your project's `settings.py` file:

```
INSTALLED_APPS = [
    ...
    'permissions_auditor',
    ...
]
```

2.2.3 Settings

PERMISSIONS_AUDITOR_PROCESSORS

This setting is used to configure the processors used to parse views for their permissions. You can add custom processors, or remove the default ones similar to Django's middleware system.

For details on each processor, see *Included Processors*.

Default:

```
PERMISSIONS_AUDITOR_PROCESSORS = [
    'permissions_auditor.processors.auth_mixins.PermissionRequiredMixinProcessor',
    'permissions_auditor.processors.auth_mixins.LoginRequiredMixinProcessor',
    'permissions_auditor.processors.auth_mixins.UserPassesTestMixinProcessor',
    'permissions_auditor.processors.auth_decorators.
↪PermissionRequiredDecoratorProcessor',
    'permissions_auditor.processors.auth_decorators.LoginRequiredDecoratorProcessor',
    'permissions_auditor.processors.auth_decorators.
↪StaffMemberRequiredDecoratorProcessor',
    'permissions_auditor.processors.auth_decorators.
↪SuperUserRequiredDecoratorProcessor',
    'permissions_auditor.processors.auth_decorators.UserPassesTestDecoratorProcessor',
]
```

PERMISSIONS_AUDITOR_BLACKLIST

Exclude views from parsing that match the blacklist values.

Default:

```
PERMISSIONS_AUDITOR_BLACKLIST = {
    'namespaces': [
        'admin',
    ],
    'view_names': [],
    'modules': [],
}
```

namespaces URL namespaces that will be blacklisted. By default, all views in the `admin` namespace are blacklisted.

view_names Fully qualified view paths to be blacklisted. Example: `test_app.views.home_page`.

modules Modules to be blacklisted. Example: `test_app.views.function_based`.

PERMISSIONS_AUDITOR_ADMIN

Enable or disable the Django admin page provided by the app. If `TRUE`, the admin site will be enabled. Useful if you want to create a custom management page instead of using the Django admin.

Default: `TRUE`

PERMISSIONS_AUDITOR_ROOT_URLCONF

The root Django URL configuration to use when fetching views.

Default: The `ROOT_URLCONF` value in your Django project's `settings.py` file.

PERMISSIONS_AUDITOR_CACHE_KEY

The cache key prefix to use when caching processed views results.

Default: `'permissions_auditor_views'`

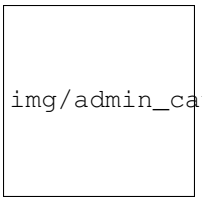
PERMISSIONS_AUDITOR_CACHE_TIMEOUT

The timeout to use when caching processed views results.

Default: `900`

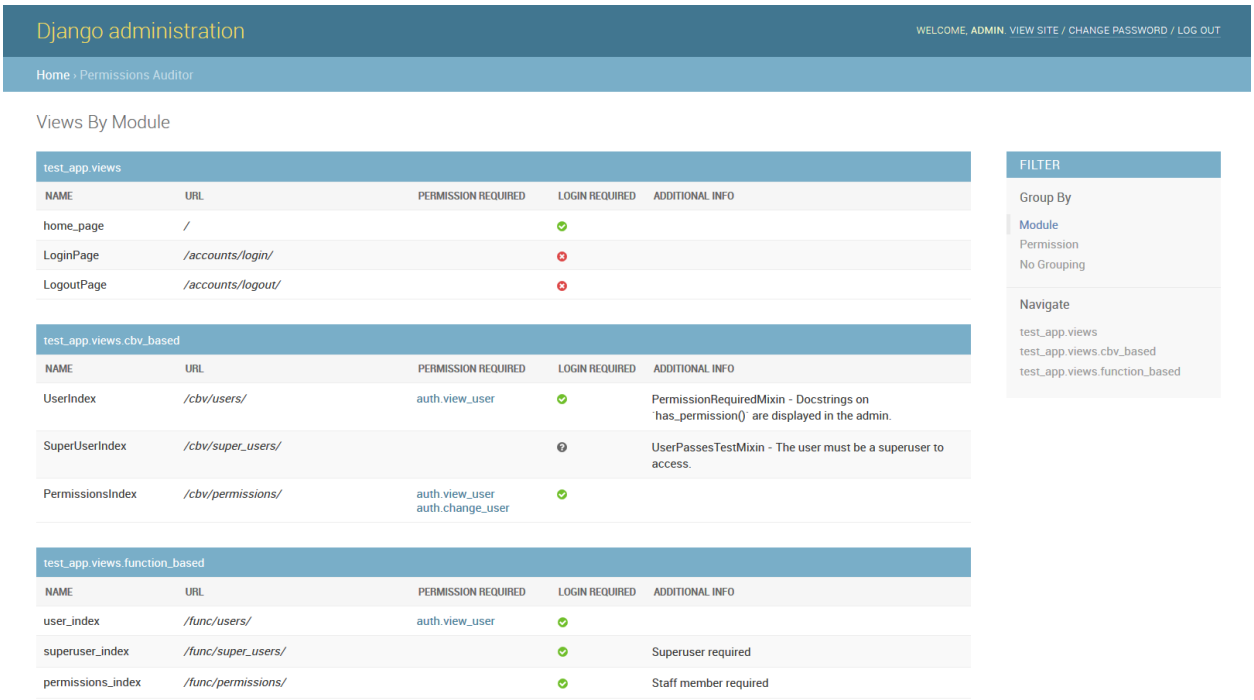
2.2.4 Admin Site

Once installed, you should see a *Permissions Auditor* category in your Django admin panel.



Note: All staff members will be able to access the index.

Views Index

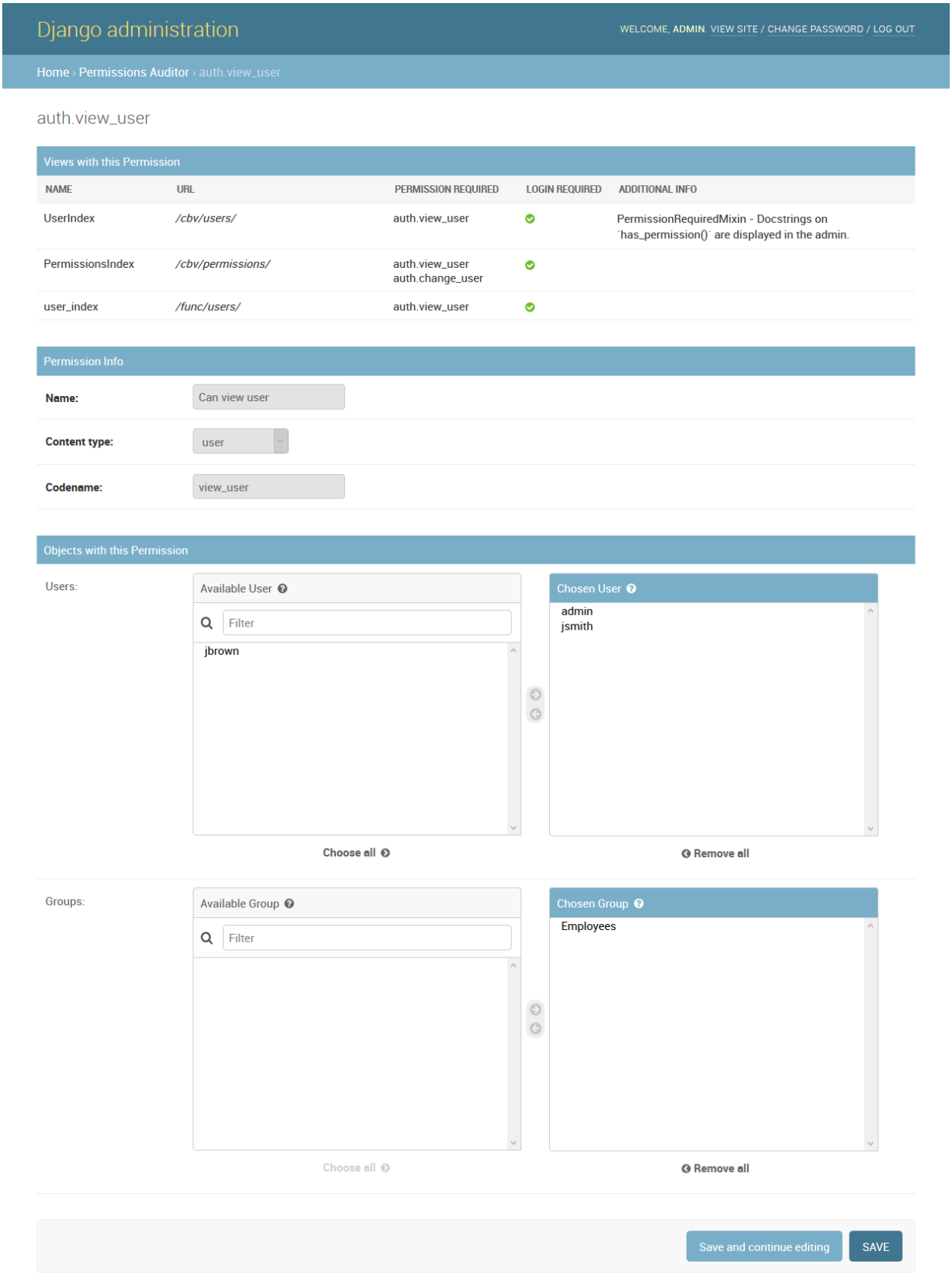


Your registered site views should display with the permissions required and any additional information in the table.

Note: If you see unexpected results, or missing permissions, ensure your *Included Processors* are correctly configured. You may need to create a custom processor if you have a view that does not use the built-in Django auth mixins / decorators.

When you click on a permission, you will be taken to a page which will allow you to manage what users and groups have that permission.

Permissions Management Page



Note: In order to modify permissions on this page, the user must have the `auth.change_user` and `auth.change_group` permissions.

2.2.5 Example Views

The following are example views are detected out of the box. For more examples, see `permissions_auditor/tests/fixtures/views.py`.

Simple Permission Required Page

Listing 1: views.py

```
from django.contrib.auth.mixins import PermissionRequiredMixin
from django.views.generic import TemplateView

class ExampleView(PermissionRequiredMixin, TemplateView):
    template_name = 'example.html'
    permission_required = 'auth.view_user'

    ...
```

Listing 2: urls.py

```
from django.urls import path
from views import ExampleView

urlpatterns = [
    path('', ExampleView.as_view(), name='example'),
]
```

Result:

Name	URL	Permission Required	Login Required	Additional Info
ExampleView	/	`auth.view_user`	True	

Custom Permission Required Page

In this example, we only want users with the first name 'bob' to be able to access the page.

Listing 3: views.py

```
from django.contrib.auth.mixins import PermissionRequiredMixin
from django.views.generic import TemplateView

class BobView(PermissionRequiredMixin, TemplateView):
    template_name = 'example.html'

    def has_permission(self):
        """
        Only users with the first name Bob can access.
```

(continues on next page)

(continued from previous page)

```
        """
        return self.request.user.first_name == 'Bob'

    ...
```

Listing 4: urls.py

```
from django.urls import path
from views import BobView

urlpatterns = [
    path('/bob/', BobView.as_view(), name='bob'),
]
```

Result:

Name	URL	Permission Required	Login Required	Additional Info
ExampleView	*/bob/*		True	Only users with the first name Bob can access.

Hint: The *PermissionRequiredMixinProcessor* will display the docstring on the the `has_permission()` function in the additional info column.

Simple Login Required View

Listing 5: views.py

```
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    ...
```

Listing 6: urls.py

```
from django.urls import path
from views import my_view

urlpatterns = [
    path('', my_view, name='example'),
]
```

Result:

Name	URL	Permission Required	Login Required	Additional Info
my_view	/		True	

2.2.6 Included Processors

Processors are what do the work of parsing the permissions out of a view.

Django Auth Decorator Processors

PermissionRequiredDecoratorProcessor

class permissions_auditor.processors.auth_decorators.**PermissionRequiredDecoratorProcessor**
Process `@permission_required()` decorator on function based views.

LoginRequiredDecoratorProcessor

class permissions_auditor.processors.auth_decorators.**LoginRequiredDecoratorProcessor**
Process `@login_required` decorator on function based views.

StaffMemberRequiredDecoratorProcessor

class permissions_auditor.processors.auth_decorators.**StaffMemberRequiredDecoratorProcessor**
Process Django admin's `@staff_member_required` decorator on function based views.

ActiveUserRequiredDecoratorProcessor

class permissions_auditor.processors.auth_decorators.**ActiveUserRequiredDecoratorProcessor**
Process `@user_passes_test(lambda u: u.is_active)` decorator on function based views.

AnonymousUserRequiredDecoratorProcessor

class permissions_auditor.processors.auth_decorators.**AnonymousUserRequiredDecoratorProcesso**
Process `@user_passes_test(lambda u: u.is_anonymous)` decorator on function based views.

SuperUserRequiredDecoratorProcessor

class permissions_auditor.processors.auth_decorators.**SuperUserRequiredDecoratorProcessor**
Process `@user_passes_test(lambda u: u.is_superuser)` decorator on function based views.

UserPassesTestDecoratorProcessor

class permissions_auditor.processors.auth_decorators.**UserPassesTestDecoratorProcessor**
Process `@user_passes_test()` decorator on function based views.

Note: the `@user_passes_test` decorator does not automatically check that the User is not anonymous. This means they don't necessarily need to be authenticated for the check to pass, so this processor returns `None` (unknown) for the login_required attribute.

Django Auth Mixin Processors

PermissionRequiredMixinProcessor

class permissions_auditor.processors.auth_mixins.**PermissionRequiredMixinProcessor**
Processes views that directly inherit from `django.contrib.auth.mixins.PermissionRequiredMixin`.

Hint: If the `has_permission()` function is overridden, any docstrings on that function will be displayed in the additional info column.

LoginRequiredMixinProcessor

class permissions_auditor.processors.auth_mixins.**LoginRequiredMixinProcessor**
Processes views that directly inherit from `django.contrib.auth.mixins.LoginRequiredMixin`.

UserPassesTestMixinProcessor

class permissions_auditor.processors.auth_mixins.**UserPassesTestMixinProcessor**
Processes views that directly inherit from `django.contrib.auth.mixins.UserPassesTestMixin`.

Hint: If the function returned by `get_test_func()` is overridden, any docstrings on that function will be displayed in the additional info column.

Note: UserPassesTestMixinProcessor does not automatically check that the User is not anonymous. This means they don't necessarily need to be authenticated for the check to pass, so this processor returns `None` (unknown) for the login_required attribute.

2.2.7 Custom Processors

Base Processors

All processors inherit from `BaseProcessor`.

class permissions_auditor.processors.base.**BaseProcessor**

can_process(*view*)
Can this processor process the provided view?

Parameters **view** (*function or class*) – the view being processed.

Returns whether this processor can process the view. Default: `False`

Return type boolean

get_docstring(*view*)
Return any additional information that should be displayed when showing permisison information.

Parameters **view** (*function or class*) – the view being processed.

Returns the string to display in the additional info column. Default: `None`

Return type str or None

get_login_required(*view*)
Get whether or not the view needs the user to be logged in to access.

Parameters **view** (*function or class*) – the view being processed.

Returns whether a user must be logged in to access this view. Default: `False`

Return type boolean or None (if unknown)

get_permission_required(*view*)
Get the permissions required on the provided view. Must return an iterable.

Parameters **view** (*function or class*) – the view being processed.

Returns the permissions required to access the view. Default: `[]`

Return type list(str)

Other useful base classes:

class permissions_auditor.processors.base.**BaseFuncViewProcessor**
Base class for processing function based views.

class permissions_auditor.processors.base.**BaseCBVProcessor**
Base class for processing class based views.

class permissions_auditor.processors.base.**BaseFileredMixinProcessor**
Base class for parsing mixins on class based views. Set `class_filter` to filter the class names the processor applies to. ONLY checks top level base classes.

Variables **class_filter** – initial value: `None`

get_class_filter()
Override this method to override the class_names attribute. Must return an iterable.

Returns a list of strings containing the full paths of mixins to detect.

Raises **ImproperlyConfigured** – if the `class_filter` atribute is `None`.

Parsing Mixins

Creating a custom processor for mixins on class based views is fairly straight forward.

In this example, we have a mixin `BobRequiredMixin` and a view that uses it, `BobsPage`. The mixin should only allow users with the first name Bob to access the page.

Listing 7: example_project/views.py

```
from django.core.exceptions import PermissionDenied
from django.views.generic import TemplateView

class BobRequiredMixin:
    def dispatch(self, request, *args, **kwargs):
        if self.request.user.first_name != 'Bob':
            raise PermissionDenied("You are not Bob")
        return super().dispatch(request, *args, **kwargs)

class BobsPage(BobRequiredMixin, TemplateView):
    ...
```

Let's define our processor in *processors.py*.

Listing 8: example_project/processors.py

```
from permissions_auditor.processors.base import BaseFileredMixinProcessor

class BobRequiredMixinProcessor(BaseFileredMixinProcessor):
    class_filter = 'example_project.views.BobRequiredMixin'

    def get_login_required(self, view):
        return True

    def get_docstring(self, view):
        return "The user's first name must be Bob to view."
```

To register our processor, we need to add it to *PERMISSIONS_AUDITOR_PROCESSORS* in our project settings.

Listing 9: settings.py

```
PERMISSIONS_AUDITOR_PROCESSORS = [
    ...

    'example_project.processors.BobRequiredProcessor',
]
```

When `BobsPage` is registered to a URL, we should see this in the admin panel:

Name	URL	Permission Required	Login Required	Additional Info
BobsPage	/		True	The user's first name must be Bob to view.

Perhaps we want to make our mixin configurable so we can detect different names depending on the view. We also have multiple people with the same first name, so we also want to check for a permission: `example.view_pages`.

```
class FirstNameRequiredMixin:
    required_first_name = ''

    def dispatch(self, request, *args, **kwargs):
        if not (self.request.user.has_perm('example_app.view_userpages')
                and self.request.user.first_name == self.required_first_name):
            raise PermissionDenied()
        return super().dispatch(request, *args, **kwargs)

class GeorgesPage(FirstNameRequiredMixin, TemplateView):
    required_first_name = 'George'

    ...
```

We'll modify `class_filter` and `get_docstring()` from our old processor, and override `get_permission_required()`.

```
from permissions_auditor.processors.base import BaseFileredMixinProcessor

class FirstNameRequiredMixinProcessor(BaseFileredMixinProcessor):
    class_filter = 'example_project.views.FirstNameRequiredMixin'

    def get_permission_required(self, view):
        return ['example.view_pages']
```

(continues on next page)

(continued from previous page)

```
    def get_login_required(self, view):
        return True

    def get_docstring(self, view):
        return "The user's first name must be {} to view.".format(view.first_name_
↪required)
```

Once we register our view to a URL and register the processor, our admin table should look like this:

Name	URL	Permission Required	Login Required	Additional Info
Georges-Page	/	example.view_pages	True	The user's first name must be George to view.

Additional Examples

See the `permissions_auditor/processors/` folder in the source code for more examples.

2.2.8 Changelog

v0.3.3 (Released 1/9/2019)

- Mark docstrings as safe in admin templates
- No longer suppress inner exceptions when parsing processors
- Fix Django admin module permissions check

v0.3.2 (Released 1/9/2019)

- Fix various cache issues
- Only show active users in the admin permission configuration page

v0.3.1 (Released 1/8/2019)

Initial stable release

Index