Django Auth Functional Documentation

Release 0.1.0

Anler

August 29, 2015

Contents

CHAPTER 1

What is this?

This library provides a set of decorators for working with authentication and authorization. These decorators can be used to decorate plain functions or method in class-based views and you can decide what http response you want to return in the cases where the authentication/authorization failed.

CHAPTER 2

Authenticating your views

In order to authenticate your views all you need to do is decorate your view function:

```
from auth_functional import authentication
from django.template.response import TemplateResponse

@authentication
def profile(request):
    return TemplateResponse(request, 'user/profile.html')
```

Or, in case you're using aa class-base view:

```
from auth_functional import authentication
from django.template.response import TemplateResponse
from django.views.generic import View

class SomeView(View):
    @authentication
    def get(self, request):
        return TemplateREsponse(request, 'user/profile.html')
```

With that in place, all the non-authenticated requests are gonna receive an **HTTP 401 Unauthorized** response.

2.1 Requesting client authentication

When you want the user agent to authenticate itself towards the server, you can send a request for authentication using the `WWW-Authenticate header`. Here's an example using basic authentication:

```
from auth_functional import authentication
from django.template.response import TemplateResponse

@authentication(www_authenticate='Basic realm="private area"')
def profile(request):
    return TemplateResponse(request, 'user/profile.html')
```

2.2 Returning a different response

If you want to return a response different than the default **HTTP 401 Unauthorized** you can provide `response_factory` callable to the authentication decorator. If the authentication fails your `response_factory` callable will be called with **the same parameters as the view**.

```
from auth_functional import authentication
from django.template.response import TemplateResponse
from django import http


def unauthorized_response(request):
    response = http.HttpResponse(status=401)
    if 'application/json' in request.META.get('HTTP_ACCEPT'):
        response['Content-Type'] = 'application/json; charset=utf-8'
    return response


@authentication(response_factory=unauthorized_response)
def profile(request):
    return TemplateResponse(request, 'user/profile.html')
```

CHAPTER 3

Authorizing your views

In order to authorize your views all you need to do is decorate your view function with the properly named `authorization` decorator passing a **condition** callable that is in charge of allowing or not the access to the resource/controller/store:

```
from auth_functional import authentication, authorization
from django.template.response import TemplateResponse


def is_staff(request):
    return request.user.is_staff


@authentication
@authorization(condition=is_staff)
def profile(request):
    return TemplateResponse(request, 'user/profile.html')
```

Or, in case you're using aa class-base view:

```
from auth_functional import authentication, authorization
from django.template.response import TemplateResponse
from django.views.generic import View


def is_staff(request):
    return request.user.is_staff


class SomeView(View):
    @authentication
    @authorization(condition=is_staff)
    def get(self, request):
        return TemplateREsponse(request, 'user/profile.html')
```

With that in place, all the non-authorized requests are gonna receive an **HTTP 403 Forbidden** response which means that the client doesn't have access.

3.1 Returning a different response

If you want to return a response different than the default **HTTP 403 Forbidden** you can provide `response_factory` callable to the authorization decorator. If the authorization fails your `response_factory`

callable will be called with **the same parameters as the view**.

```
from auth_functional import authentication, authorization
from django.template.response import TemplateResponse
from django import http


def forbidden_response(request):
    response = http.HttpResponse(status=403)
    if 'application/json' in request.META.get('HTTP_ACCEPT'):
        response['Content-Type'] = 'application/json; charset=utf-8'
    return response


def is_staff(request):
    return request.user.is_staff


@authentication
@authorization(condition=is_staff, response_factory=forbidden_response)
def profile(request):
    return TemplateResponse(request, 'user/profile.html')
```

3.2 Combining multiple conditions

You can combine different condition callables by using the `and_`, `or_` and `not_` decorators:

```
from auth_functional import authentication, authorization, and_, not_
from django.template.response import TemplateResponse
from django import http


def forbidden_response(request):
    response = http.HttpResponse(status=403)
    if 'application/json' in request.META.get('HTTP_ACCEPT'):
        response['Content-Type'] = 'application/json; charset=utf-8'
    return response


def is_staff(request):
    return request.user.is_staff

def is_admin(request):
    return request.user.is_admin

@authentication
@authorization(condition=and_(is_staff, _not(is_admin)), response_factory=forbidden_response)
def profile(request):
    return TemplateResponse(request, 'user/profile.html')
```

CHAPTER 4

Improving performance by using request cache

When using multiple conditions you may end repeating operations to fetch the objects and check permissions. Let's say you have the following two conditions, one that checks the user is the owner of a video, and the other that the user can play the video (some sort of premium feature, whatever):

```
from auth_functional import authentication, authorization, and_
from django.template.response import TemplateResponse
from django import http
from myapp.models import Video


def can_download_video(request, video_id):
    video = Video.objects.filter(pk=video_id).get()
    return video.user == request.user


def can_play_video(request, video_id):
    video = Video.objects.filter(pk=video_id).get()
    return video.has_be_played_by(request.user)


@authentication
@authorization(condition=and_(is_owner_of_video, can_play_video)
def play_video(request, video_id):
    return TemplateResponse(request, 'user/video.html')
```

As you can see, you ended up fetching the same video twice from the database `Video.objects.filter(pk=video_id).get()`. You could create another condition that checks both conditions but that would miss the point of creating smaller conditions and combine them with `and_` avoiding the conditions to be too coupled with the logic of your app.

A workaround to avoid this duplicated logic to fetch a needed objects is to use the `request.fixture` to fetch and cache the object for the current request:

```
from auth_functional import authentication, authorization, and_
from django.template.response import TemplateResponse
from django import http
from myapp.models import Video

auth_functional.install_request_fixture('video', Video.objects.get)


def can_download_video(request, video_id):
    return request.fixture.video(pk=video_id).user == request.user
```

```
def can_play_video(request, video_id):
    return request.fixture.video(pk=video_id).has_be_played_by(request.user)


@authentication
@authorization(condition=and_(is_owner_of_video, can_play_video)
def play_video(request, video_id):
    return TemplateResponse(request, 'user/video.html')
```

With that in place the first time you access the fixture your registered callable is going to be called and the result will be cached **but only during the current request lifetime**. This way any subsequent call to the fixture will return only the cached value.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search