

# 认证、权限和限制

---

简单来说就是：

认证确定了你是谁

权限确定你能不能访问某个接口

限制确定你访问某个接口的频率

## 一、认证

---

身份验证是将传入请求与一组标识凭据（例如请求来自的用户或其签名的令牌）相关联的机制。然后权限和限制 组件决定是否拒绝这个请求。**认证本身不会允许或拒绝传入的请求**，它只是简单识别请求携带的凭证。

REST framework 提供了一些开箱即用的身份验证方案，并且还允许你实现自定义方案。

```
# 基于用户名和密码的认证
+ class BasicAuthentication(BaseAuthentication):...

# 基于Session的认证
+ class SessionAuthentication(BaseAuthentication):...

# 基于Token的认证
+ class TokenAuthentication(BaseAuthentication):...

# 基于远端用户的认证（专用用户管理服务器）
+ class RemoteUserAuthentication(BaseAuthentication):...
```

### 1.自定义认证

要实现自定义的认证方案，要继承BaseAuthentication类并且重写.authenticate(self, request) 方法。如果认证成功，该方法应返回(user, auth)的二元元组，否则返回None。

在某些情况下，你可能不想返回None，而是希望从.authenticate()方法抛出AuthenticationFailed异常。

在App下，创建authentications.py，然后自定义认证类

```
from django.core.cache import cache
from rest_framework.authentication import BaseAuthentication
from rest_framework.exceptions import AuthenticationFailed # 用于抛出
错误信息

from App.models import User

class MyAuthentication(BaseAuthentication):
    def authenticate(self, request):
        token = request.query_params.get('token')
        if not token:
            raise AuthenticationFailed("没有token")
        uid = cache.get(token)
        if not uid:
            raise AuthenticationFailed("token过期")
        try:
            user = User.objects.get(pk=uid)
            if user:
                return user, None
            else:
                raise AuthenticationFailed("token不存在")
        except Exception as e:
            raise AuthenticationFailed("token不合法")
```

## 2.全局级别认证配置

在settings中配置

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'App.authentications.MyAuthentication',
    )
}
```

### 3.视图级别认证配置

```
from App.authentications import MyAuthentication
from rest_framework.generics import GenericAPIView

class ExampleView(GenericAPIView):
    authentication_classes = (MyAuthentication, )
    ...
```

### 4. 实现一个自定义认证方案

- 定义一个用户模型，访问用户信息

```
class User(models.Model):
    username = models.CharField(unique=True, max_length=150)
    password = models.CharField(max_length=128)
    usertype = models.IntegerField(choices=((1, '超管'), (2, '管理员'),
    (3, '普通用户')), default=3)
    email = models.CharField(max_length=254, null=True)
    date_joined =
models.DateTimeField(default=datetime.now, null=True)

    class Meta:
        db_table = 'user'
```

- 用户序列化模型

```
class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = "__all__"
```

- 实现用户注册

```

from rest_framework.generics import CreateAPIView,GenericAPIView
from rest_framework.response import Response

class RegisterView(CreateAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    def post(self, request, *args, **kwargs):
        request.data._mutable = True # 修改data中的数据
        request.data["password"] =
make_password(request.data["password"])
        return self.create(request, *args, **kwargs)

```

- 实现登录

```

import uuid

from django.contrib.auth.hashers import make_password, check_password
from django.core.cache import cache
from rest_framework.generics import
ListAPIView,CreateAPIView,GenericAPIView
from rest_framework.response import Response
class UserLoginView(GenericAPIView):
    authentication_classes = (MyAuthentication, )
    def post(self, request, *args, **kwargs):
        username = request.data.get('username')
        password = request.data.get('password','')
        user = User.objects.filter(username=username).first()
        if user and check_password(password,user.password):
            # 产生token
            token = uuid.uuid4().hex
            # 写入缓存
            cache.set(token,user.id,3600)
            return Response({'code':1,'msg':'登录成功','token':token})
        else:
            return Response({'code': -1, 'msg': '用户名或密码错'})

```

## 二、权限控制

权限控制可以限制用户对于视图的访问和对于具体数据对象的访问。

- 在执行视图的dispatch()方法前，会先进行视图访问权限的判断
- 在通过get\_object()获取具体对象时，会进行对象访问权限的判断

rest\_framework也给我提供了相应的支持，接下来看下他的原码：

```
class BasePermission(object):
    """
    A base class from which all permission classes should inherit.
    """
    #判断是否对视图有权限
    def has_permission(self, request, view):
        """
        Return `True` if permission is granted, `False` otherwise.
        """
        return True
    判断是否对某个模型有权限
    def has_object_permission(self, request, view, obj):
        """
        Return `True` if permission is granted, `False` otherwise.
        """
        return True
```

## 系统内置的权限

- AllowAny 允许所有用户
- IsAuthenticated 仅通过认证的用户
- IsAdminUser 仅管理员用户
- IsAuthenticatedOrReadOnly 认证的用户可以完全操作，否则只能get读取

## 代码实现

如果我们想编写一个权限的控制

- 首先，我们要在app目录下新建一个 文件比如是permission.py，然后自定义一个权限类

```

from rest_framework.permissions import BasePermission
#写一个继承自BasePermission
class SuperPerssion(BasePermission):
    #复写里面的has_permission函数
    def has_permission(self, request, view):
        #判断当前用户是不是超级管理员
        return request.user.is_superuser

class StaffPerssion(BasePermission):

    def has_permission(self, request, view):
        return request.user.is_staff

```

- 对应的Serializer

```

from django.contrib.auth.models import User
from rest_framework import serializers
from .models import Book

class UsersSerializer(serializers.ModelSerializer):

    class Meta:
        model = User
        fields = ("id", "username", "email")

```

- 视图级别权限认证

```

class UsersAPI(ListAPIView):
    queryset = User.objects.all()
    #指定使用的序列化器
    serializer_class = UsersSerializer
    authentication_classes = (MyAuthentication, )
    #权限控制
    permission_classes = (SuperPerssion, StaffPerssion)
    def list(self, request, *args, **kwargs):
        user = request.user
        if not isinstance(user, User):
            return Response({
                "code": LOGIN_FAIL_1_CODE,
                "msg": "用户名或密码错误"
            })

```

```
queryset = self.filter_queryset(self.get_queryset())

page = self.paginate_queryset(queryset)
if page is not None:
    serializer = self.get_serializer(page, many=True)
    return self.get_paginated_response(serializer.data)

serializer = self.get_serializer(queryset, many=True)
return Response(serializer.data)
```

- 全局级别的权限认证

在settings中配置

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'App.authentications.MyAuthentication',
    ),
    "DEFAULT_PERMISSION_CLASSES":
    ["app.permission.SuperPerssion", "app.permission.StaffPerssion" ]
}
```

## 三、节流器

---

节流类似于权限，用于控制客户端可以对API发出的请求的速率。

- 自定义类继承内置类

```
#在app下自定义mythrottle.py
from rest_framework.throttling import SimpleRateThrottle
class VisitThrottle(SimpleRateThrottle):
    #转换频率每分钟5次, 转换频率 = num/duration,其中duration可以是s(秒)、
    m(分)、h(小时)、d(天)
    rate = '5/m'
    score='vistor'
    #返回一个唯一标示用以区分不同的用户
    def get_cache_key(self, request, view):
        return self.get_ident(request) #返回用户ip
```

- 局部限制

```
#视图类
# 获取当前用户创建的书籍 要包括用户信息和他所有相关书籍的数据
class xxxAPIView(ListAPIView):
    throttle_classes = (VisitThrottle,)
    ...
```

- 全局限制

```
#在settings中设置
'DEFAULT_THROTTLE_CLASSES': ['apps.mythrottle.VisitThrottle'],

'DEFAULT_THROTTLE_RATES': {
    'vistor': '3/m',
    'anon': '100/day',# 匿名用户
},
```

`DEFAULT_THROTTLE_RATES` 可以使用 `second`, `minute`, `hour` 或 `day` 来指明周期。

## 可选限流类

### 1) AnonRateThrottle

限制所有匿名未认证用户, 使用IP区分用户。

使用 `DEFAULT_THROTTLE_RATES['anon']` 来设置频次



## 2) UserRateThrottle

限制认证用户，使用User id 来区分。

使用 `DEFAULT_THROTTLE_RATES['user']` 来设置频次

## 3) ScopedRateThrottle

限制用户对于每个视图的访问频次，使用ip或user id。

例如：

```
class ContactListView(APIView):
    throttle_scope = 'contacts'
    ...

class ContactDetailView(APIView):
    throttle_scope = 'contacts'
    ...

class UploadView(APIView):
    throttle_scope = 'uploads'
    ...

REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': (
        'rest_framework.throttling.ScopedRateThrottle',
    ),
    'DEFAULT_THROTTLE_RATES': {
        'contacts': '1000/day',
        'uploads': '20/day'
    }
}
```

# 四、分页Pagination

---

REST framework提供了分页的支持。

我们可以在配置文件中设置全局的分页方式，如：

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
    'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 10 # 每页数目
}
```

也可通过自定义Pagination类，来为视图添加不同分页行为。在视图中通过 `pagination_class` 属性来指明。

```
class LargeResultsSetPagination(PageNumberPagination):
    page_size = 1000
    page_size_query_param = 'page_size'
    max_page_size = 10000
class BookDetailView(RetrieveAPIView):
    queryset = BookInfo.objects.all()
    serializer_class = BookInfoSerializer
    pagination_class = LargeResultsSetPagination
```

注意：如果在视图内关闭分页功能，只需在视图内设置

```
pagination_class = None
```

## 可选分页器

### 1) PageNumberPagination

前端访问网址形式：

```
GET http://api.example.org/books/?page=4
```

可以在子类中定义的属性：

- `page_size` 每页数目
- `page_query_param` 前端发送的页数关键字名，默认为"page"
- `page_size_query_param` 前端发送的每页数目关键字名，默认为None
- `max_page_size` 前端最多能设置的每页数量

```

from rest_framework.pagination import PageNumberPagination

class StandardPageNumberPagination(PageNumberPagination):
    page_size_query_param = 'page_size'
    max_page_size = 10

class BookListView(ListAPIView):
    queryset = BookInfo.objects.all().order_by('id')
    serializer_class = BookInfoSerializer
    pagination_class = StandardPageNumberPagination

# 127.0.0.1/books/?page=1&page_size=2

```

## 2) LimitOffsetPagination

前端访问网址形式：

```
GET http://api.example.org/books/?limit=100&offset=400
```

可以在子类中定义的属性：

- default\_limit 默认限制，默认值与 `PAGE_SIZE` 设置一直
- limit\_query\_param limit参数名，默认'limit'
- offset\_query\_param offset参数名，默认'offset'
- max\_limit 最大limit限制，默认None

```

from rest_framework.pagination import LimitOffsetPagination

class BookListView(ListAPIView):
    queryset = BookInfo.objects.all().order_by('id')
    serializer_class = BookInfoSerializer
    pagination_class = LimitOffsetPagination

# 127.0.0.1:8000/books/?offset=3&limit=2

```

# 自动生成接口文档

REST framework可以自动帮助我们生成接口文档。

接口文档以网页的方式呈现。

自动接口文档能生成的是继承自 `APIView` 及其子类的视图。

## 1. 安装依赖

REST framewrok生成接口文档需要 `coreapi` 库的支持。

```
pip install coreapi
```

## 2. 设置接口文档访问路径

在总路由中添加接口文档路径。

文档路由对应的视图配置为 `rest_framework.documentation.include_docs_urls`,

参数 `title` 为接口文档网站的标题。

```
from rest_framework.documentation import include_docs_urls

urlpatterns = [
    ...
    url(r'^docs/', include_docs_urls(title='My API title'))
]
```

## 3. 文档描述说明的定义位置

1) 单一方法的视图, 可直接使用类视图的文档字符串, 如

```
class BookListView(generics.ListAPIView):
    """
    返回所有图书信息.
    """
```

2) 包含多个方法的视图, 在类视图的文档字符串中, 分开方法定义, 如

```
class BookListCreateView(generics.ListCreateAPIView):
    """
    get:
    返回所有图书信息.

    post:
    新建图书.
    """
```

3) 对于视图集ViewSet，仍在类视图的文档字符串中分开定义，但是应使用action名称区分，如

```
class BookInfoViewSet(mixins.ListModelMixin,
mixins.RetrieveModelMixin, GenericViewSet):
    """
    list:
    返回图书列表数据

    retrieve:
    返回图书详情数据

    latest:
    返回最新的图书数据

    read:
    修改图书的阅读量
    """
```

## 4 配置

在settings中配置

```
REST_FRAMEWORK = {
    'DEFAULT_SCHEMA_CLASS':
    'rest_framework.schemas.coreapi.AutoSchema'
}
```

## 5. 访问接口文档网页

浏览器访问 127.0.0.1:8000/docs/，即可看到自动生成的接口文档。

My API title

Index

login

register

Authenticationnone

Source Codeshell

# My API title

## index

### list

GET/index/

Interact

首页

# Load the schema document  
\$ coreapi get http://127.0.0.1:9909/docs/  
  
# Interact with the API endpoint  
\$ coreapi action index list

### login

### create

POST/login/

Interact

用户登录

**Request Body**  
The request body should be a "application/json" encoded object, containing the following items.

Parameter	Description
username	required

# Load the schema document  
\$ coreapi get http://127.0.0.1:9909/docs/  
  
# Interact with the API endpoint  
\$ coreapi action login create -p username=... -p password=...

## 两点说明：

- 1) 视图集ViewSet中的retrieve名称，在接口文档网站中叫做read
- 2) 参数的Description需要在模型类或序列化器类的字段中以help\_text选项定义，如：

```
class BookInfo(models.Model):  
    ...  
    bread = models.IntegerField(default=0, verbose_name='阅读量',  
    help_text='阅读量')  
    ...
```

或

```
class BookReadSerializer(serializers.ModelSerializer):  
    class Meta:  
        model = BookInfo  
        fields = ('bread', )  
        extra_kwargs = {  
            'bread': {  
                'required': True,  
                'help_text': '阅读量'  
            }  
        }  
    }
```

