一、验证器Validators

在Django REST framework 序列化器中进行验证和 在Django `ModelForm` 类上进行验证有一点的点区别.

对于 `ModelForm` ，一部分验证在表单上执行，一部分在模型实例上执行。而使用REST框架时，验证完全在序列化类上执行。 它有下面的优点：

- 它引入了适当的关注点分离，使代码行为更加明显。
- 在使用快捷方式ModelSerializer类和使用显式的序列化类之间切换很容易。用于ModelSerializer的任何验证行为都很容易复制。
- 打印序列化程序实例可以显示它应用的验证规则。没有对模型实例调用额外的隐藏的验证行为。

当你使用（继承） `ModelSerializer` 类的时候，所有的验证动作都是自动的。而如果你使用（继承）更底层的 `Serializer` 类的时候，你需要显式地定义验证器地规则。

我们以下面的模型为例子，来演示如何在DRF中进行验证工作，这个模型有一个字段是unique的：

```
class CustomerReportRecord(models.Model):
    time_raised = models.DateTimeField(default=timezone.now, editable=False)
    reference = models.CharField(unique=True, max_length=20) # 注意这个字段
    description = models.TextField()
```

下面则是一个序列化类，继承了ModelSerializer:

```
class CustomerReportSerializer(serializers.ModelSerializer):
    class Meta:
        model = CustomerReportRecord
        fields = '__all__'
```

使用 `python manage.py shell` 进入shell环境，尝试下面的操作：

```
>>> from project.example.serializers import CustomerReportSerializer
>>> serializer = CustomerReportSerializer()
>>> print(repr(serializer))
CustomerReportSerializer():
    id = IntegerField(label='ID',  read_only=True)   # 会自动生成这个字段，并且只读
    time_raised = DateTimeField(read_only=True)
    reference = CharField(max_length=20, validators=[<UniqueValidator(queryset=CustomerReportRecord.objects.all())>])
    description = CharField(style={'type': 'textarea'})
```

注意其中的 `reference` 字段，它被显式地指定了一个unique验证。

REST framework包含了一系列验证器类，这些类不属于Django核心原生的，它们都位于DRF的validators模块里。

UniqueValidator

用于对 `unique=True` 约束进行验证的验证器。 主要参数如下：

- `queryset` 必填参数 - 需要满足unique约束的查询集。
- `message` - 可选参数。当验证失败时自定义的错误信息。
- `lookup` - 查询的匹配方法或者说匹配模式，默认是`'exact'`。

看下面的例子：

```
from rest_framework.validators import  UniqueValidator

slug =
    SlugField( max_length=100,
    validators=[UniqueValidator(queryset=BlogPost.objects.all())]
)
```

UniqueTogetherValidator

用于 `unique_together` 联合唯一的验证。参数如下：

- `queryset` 必填参数 - 验证器应用的查询集

- `fields` 必填参数 - 一个字段名的列表或元组，它们需要成为联合唯一不重复的集合，必须是序列化类中的字段。

- `message` - 可选参数。当验证失败时自定义的错误信息。在序列化类中，可以参考下面的使用方式

```
from rest framework.validators import UniqueTogetherValidator

class ExampleSerializer(serializers.Serializer):
    # ...
    class Meta:
        validators =
            [ UniqueTogetherValidator(
                queryset=ToDoItem.objects.all(),
                fields=('list', 'position')
            )
        ]
```

UniqueForDateValidator、UniqueForMonthValidator、UniqueForYearValidator

以上三个验证器用于验证 `unique_for_date`，`unique_for_month` 和 `unique_for_year` 约束。参数如下：

- `queryset` 同上，必填参数。
- `field` 必填参数，实施验证的字段，字段的值必须在要求的时间范围内。
- `date_field` 必填参数-用于决定时间范围的字段。
- `message` - 错误信息，可选。

参考范例：

```
from rest_framework.validators import UniqueForYearValidator

class ExampleSerializer(serializers.Serializer):
    # ...
    class Meta:
        # Blog posts should have a slug that is unique for the current year.
        validators =
            [ UniqueForYearValidator(
                queryset=BlogPostItem.objects.all(),
                field='slug',
                date_field='published'
            )
        ]
```

二、高级的字段默认值

在序列化时，跨多个字段应用的验证程序有时可能需要API客户端不应提供的字段输入，但可作为验证程序的输入 使用。

两种可能的应用场景：

- 使用 `HiddenField` 字段。这种字段会出现在 `validated_data` 中，但是不会用在序列化的输出表示中。
- 使用带有 `read_only=True` 属性的标准字段，但是它同时还带有 `default=…` 参数。此时，该字段将用于序列化的输出表示，但无法被用户直接设置。

REST框架提供了一些在此情景中可能有用的默认值。

- CurrentUserDefault

用于表示当前用户的默认类。为了使用它，在实例化序列化类时，“request”必须作为上下文字典的一部分提供。

```
owner = serializers.HiddenField(
default=serializers.CurrentUserDefault() 3 )
```

- CreateOnlyDefault

可用于设置的默认类，只能在创建对象的期间设置默认参数。在更新期间，该字段将被省略。

它只接收一个参数，也就是创建对象时使用的默认值或可调用对象。

```
created_at = serializers.DateTimeField(
default=serializers.CreateOnlyDefault(timezone.now) )
```

三、移除默认的验证器

在某些不明确的情况下，可能需要显式处理验证过程，而不是依赖“modelserializer”类生成的默认序列化类。在这些情况下，可以通过将序列化类的'meta.validators'属性指定为空列表来禁用自动生成的验证器。

例如：

```
class BillingRecordSerializer(serializers.ModelSerializer):
    def validate(self, data):
        #在这里自定义验证过程，或者在视图中。

    class Meta:
        fields = ('client', 'date', 'amount')
        extra_kwargs = {'client': {'required':  False}}
        validators = []   # 移除默认的验证器
```

四、自定义验证器

你可以使用任何Django原生的验证器或者自定义一个验证器。

- 基于函数的验证器

看下面的例子，当验证失败的时候，需要弹出 `serializers.ValidationError` 异常：

```
def even_number(value):
    if value % 2 != 0:
        raise serializers.ValidationError('This field must be an even number.')
```

- 也可以通过为 `Serializer` 基于类 `.validate_<field_name>` 的验证器

的子类添加	方法，自定义字段级别的

使用 `__call__` 方法编写基于类的验证器。

```
class  MultipleOf(object):
    def_init_(self,  base):
        self.base = base

    def_call_(self, value):
        if value % self.base != 0:
            message = 'This field must be a multiple of %d.' % self.base
            raise serializers.ValidationError(message)
```

在编写验证器的过程中，有时候需要使用一些上下文环境，可以在基于类的验证器中声明一个 `set_context` 方法来实现这一目的。

```
def set_context(self, serializer_field):
    # Determine if this is an update or a create  operation.
    # In ` call ` we can then use that information to modify the
validation behavior.
    self.is_update = serializer_field.parent.instance is not  None
```

一、认证

身份验证是将传入的请求与一组鉴别凭据（例如请求来自的用户或与其签名的令牌）关联的机制。然后，可以使用权限和限流策略来确定是否允许请求进入。

DRF框架提供了许多现成的身份验证方案，还允许你实现自定义的方案。

在权限和限流检查发生之前，以及在执行任何其他代码之前，始终在视图的最开始处运行身份验证。也就是说，认证过程优先级最高，最先被执行。

`request.user` 属性通常设置为 `contrib.auth` 包的 `user` 类的实例。这是Django原生的做法，请参考Django教程。

`request.auth` 属性用于任何其他附加的身份验证信息，例如，它可以用来表示请求中携带的身份验证令牌。

上面两个位于request中的属性非常重要，是认证和权限机制的核心数据。

认证的机制

允许使用的认证模式一般以一个类的列表的配置形式存在。DRF会尝试使用列表中的每个类进行认证，并使用第一个成功通过的验证类的返回值设置 `request.user` 和 `request.auth` 。

如果所有认证类尝试了一遍，但还是没有通过验证。 `request.user` 将被设置为 `django.contrib.auth.models.AnonymousUser` 的实例，也就是匿名用户。 `request.auth` 也将被设置为 `None` 。可见，DRF**的认证机制依赖**Django**的**auth**框架！**

未认证用户请求过程中， `request.user` 和 `request.auth` 的值可以通过 `UNAUTHENTICATED_USER` and `UNAUTHENTICATED_TOKEN` 这两个配置项进行修改。

认证功能的核心源代码在这里：

```
# 位于request.py模块中

def _authenticate(self):
        """
        使用每个认证类的实例来验证请求
        """
        for authenticator in self.authenticators:    # 循环指定的所有认证类
            try: # 抓取异常
```

```
                    #   执行认证类的authenticate方法，返回的是一个（user, auth）元组
                    user_auth_tuple = authenticator.authenticate(self)
                except exceptions.APIException:  # 如果认证失败，或发生错误
                    self._not_authenticated()  # 执行认证失败后的方法，用于处理后事
                    raise  #继续向上抛出异常
                #  一旦某个认证类通过了验证，返回了正常的值，那么进入结束流程
                if user_auth_tuple is not None:
                    self._authenticator = authenticator  # 将通过认证的认证类保存下来，备查
                    self.user, self.auth = user_auth_tuple  # 将返回值赋值给request对象的user和auth属性
                    return  # 结束方法，啥都不返回，因为上面一行已经将我们需要的值保存起来了。

            self._not_authenticated()  # 如果走完整个for循环都没有认证成功，进入后事处理阶段
```

配置认证方案

通过 `DEFAULT_AUTHENTICATION_CLASSES` 配置项，可以进行全局性认证方案设置，例如：

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.BasicAuthentication',
    'rest_framework.authentication.SessionAuthentication', 5 )
}
```

对于基于类的 `APIView` 视图，也可以设置视图或视图集级别的认证方案，这种粒度更细。

```
from rest_framework.authentication import SessionAuthentication, BasicAuthentication
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response
from rest_framework.views import APIView

class ExampleView(APIView):
    authentication_classes = (SessionAuthentication, BasicAuthentication)
    permission_classes = (IsAuthenticated,)

```

```
10      def get(self, request, format=None):
11          content = {
12              'user': unicode(request.user),  # `django.contrib.auth.User`
    instance.
13              'auth': unicode(request.auth),  # None
14          }
15          return Response(content)
```

对于使用 `@api_view` 装饰器转化来的视图，也可以指定视图级别的认证方案：

```
1   @api_view(['GET'])
2   @authentication_classes((SessionAuthentication, BasicAuthentication))
3   @permission_classes((IsAuthenticated,))
4   def example_view(request, format=None):
5       content =
{ 6         'user': unicode(request.user),  # `django.contrib.auth.User`
    instance.
7           'auth': unicode(request.auth),  # None
8       }
9       return Response(content)
```

如果想要某个视图不使用认证功能，可以设置：

```
1   authentication_classes = []
```

未认证和拒绝响应

当未经身份验证的请求被拒绝时，可能有两个不同的错误代码。

- HTTP 401 Unauthorized
- HTTP 403 Permission Denied

HTTP 401响应一般会包含一个 `WWW-Authenticate` 头部属性，用于指引用户如何认证。而HTTP 403响应则不会包含这个 `WWW-Authenticate` 属性。

具体产生哪种响应类型取决于身份认证方案。虽然同时可以使用多个身份认证方案，但最终只能使用一个方案来确定响应类型。**根据视图上设置的第一个身份认证类确定响应类型。**

请注意，当请求认证成功，但仍被拒绝执行时，无论身份验证方案如何，都将返回 `403 permission denied` 响应。

一切未通过认证的情况，都执行的是下面的源代码：

```
    # 位于request.py

    def _not_authenticated(self):
        """
        默认值为AnonymousUser 和 None.
        """
        self._authenticator = None   # request中指示认证器未None，也就是没有任何一个认证通过

        if  api_settings.UNAUTHENTICATED_USER:   # 如果在settings中配置了这个参数，使用它
            self.user = api_settings.UNAUTHENTICATED_USER() #默 认 叫做
 AnonymousUser
            self.user = None   # 否则设置user的值被设置为None

        if  api settings.UNAUTHENTICATED TOKEN:   #同 上
            self.auth = api_settings.UNAUTHENTICATED_TOKEN() # 默认就是None
        else:
            self.auth = None
```

Apache+mod_wsgi 模式下的专用配置

注意，如果你使用 `Apache + mod_wsgi` 的方式部署项目，认证的头部字段将不会通过WSGI程序传递，它默认是通过Apache处理，而不是应用级别。

在Apache中，使用非会话的认证机制时，你需要显式地配置mode_wsgi，用于传递必须的头部信息。也就是进行如下地配置：

```
# this can go in either server config, virtual host, directory or .htaccess
WSGIPassAuthorization On
```

二、API参考

BasicAuthentication

DRF的认证模块非常简单，只提供了几个简单的认证类，主要还是依托Djangoyuans的auth认证框架。

BasicAuthentication使用HTTP基本的认证机制，通过用户名/密码的方式验证。它通常只适用于测试工作，尽量不要用于生成环境。

用户名和密码必须在HTTP报文头部，为 `Authorization` 属性提供值为 `Basic amFjazpmZWl4dWVsb3ZlMTAw` 的方式提供。其中 `Basic` 字符串是键，后面的一串乱码是通过 `base64` 库使用明文的用户名和密码计算出的密文，这一部分工作可以通过postman工具进行。

如果认证成功， `BasicAuthentication` 提供下面的属性：

- `request.user` ：设置为一个Django的 `User` 类的实例
- `request.auth` ：设置为None

未认证成功，将响应 `HTTP 401 Unauthorized` ，并携带下面的头部信息：

```
WWW-Authenticate: Basic realm="api"
```

可以看看它的核心方法authenticate的源代码：

```
    def authenticate(self, request):
        """
        使用HTTP的基本authentication属性，提供正确的用户名和密码，并返回一个user。否则返回None。
        """
        auth = get_authorization_header(request).split() # 从http报头读取密文，并分割字符串

        if not auth or auth[0].lower() != b'basic':  #如果分割后的第一部分不是以basic开头
            return None  # 认证失败

        if len(auth) == 1:  # 如果只有一个部分，说明没有提供用户名和密码部分，认证失败
            msg = _('Invalid basic header. No credentials provided.')
            raise exceptions.AuthenticationFailed(msg)
        elif len(auth) > 2: # 如果分割出了2个以上的部分，说明格式不对，空格太多，认证失败
            msg = _('Invalid basic header. Credentials string should not contain spaces.')
```

```
15                 raise exceptions.AuthenticationFailed(msg)
16             # 只能分割成2个部分
17             try:#获取第二个部分，用base64库进行解码
18                 auth_parts =
    base64.b64decode(auth[1]).decode(HTTP_HEADER_ENCODING).partition(':')
19             except (TypeError, UnicodeDecodeError, binascii.Error):
20                 msg = _('Invalid basic header. Credentials not correctly base64
    encoded.')
21                 raise exceptions.AuthenticationFailed(msg)
22             # 拿到明文的用户名和密码
23             userid, password = auth_parts[0], auth_parts[2]
24             # 进行密码比对，返回认证结果
25             return self.authenticate_credentials(userid, password,  request)
```

TokenAuthentication（讲项目时讲）

TokenAuthentication是一种简单的基于令牌的HTTP认证。它适用于CS架构，例如普通的桌面应用程序或移动客户端。

要使用 `TokenAuthentication` 模式，你需要先配置认证类，并将 `rest_framework.authtoken`添 加到 `INSTALLED_APPS` 中，如下所示:

```
1   INSTALLED_APPS = (
2       ...
3       'rest_framework.authtoken'
4   )
5
6   REST_FRAMEWORK =
7       { 'DEFAULT_AUTHENTICATION_CLASSES':  (
8           ...
9           'rest_framework.authentication.TokenAuthentication',
10      )
11  }
12
```

注意:配置完成后，你需要运行 `python manage.py migrate` 命令，因为

`rest_framework.authtoken` 实际上是一个app或者说第三方模块，需要在数据库中生成它工作用的数据表。

接下来，你需要为你的用户创建令牌：

```
from rest framework.authtoken.models import  Token

token = Token.objects.create(user=...)
print(token.key)
```

对于要进行身份验证的客户端，令牌密钥应包含在 `authorization` HTTP头部属性中。键应该以字符串 `"token"` 作为前缀，用空格分隔两个字符串。例如：

```
Authorization: Token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b
```

上面的工作也可以在postman中进行。

注意：如果你想使用一个不同的头部关键字，比 `Bearer` ，只需要简单地继承 `TokenAuthentication` 类，并设置 `keyword` 这个类变量的值为 `Bearer` 。

成功认证后 `TokenAuthentication` 提供下面的属性：

- `request.user` ：设置为一个Django的 `User` 类的实例
- `request.auth` ：设置为一个 `rest_framework.authtoken.models.Token` 的实例。

不成功将返回 `HTTP 401 Unauthorized` 响应，并携带下面的HTTP头部信息：

```
WWW-Authenticate: Token
```

可以使用 `curl` 命令行工具测试令牌认证API ：

```
curl -X GET http://127.0.0.1:8000/api/example/ -H 'Authorization: Token
9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b'
```

那么如何为用户生成令牌呢?主要有以下几种方式：

- **通过信号机制生成令牌**

如果你希望每个用户都有一个自动生成的令牌，可以简单地捕获用户的 `post_save` 信号。这个信号是Django原生为我们提供的，请参考Django教程相关内容。

```
from django.conf import settings
from django.db.models.signals import post_save
from django.dispatch import receiver
from rest_framework.authtoken.models import  Token

@receiver(post_save, sender=settings.AUTH_USER_MODEL)
def create_auth_token(sender, instance=None, created=False, **kwargs):
    if created:
        Token.objects.create(user=instance)
```

如果你已经创建了一些用户，可以通过下面的方式为已创建的用户生成令牌：

```
from django.contrib.auth.models import User
from rest framework.authtoken.models import  Token

for user in User.objects.all():
    Token.objects.get or create(user=user)
```

- **提供获取令牌的API服务**

用户从客户端使用用户名和密码，往提供令牌服务的API发送表单或json数据。验证通过后，API将用户的令牌以json格式返回给客户端。 DRF**提供了一个内置的视图** `obtain_auth_token` **用于实现这一功能！**

首先在你的URLconf中添加下面的路由:

```
from rest_framework.authtoken import views
urlpatterns += [
path('api-token-auth/', views.obtain_auth_token), 4
]
```

路由的匹配字符串可以随意指定。

`obtain_auth_token` 视图会返回一个JSON响应，当用户名和密码通过验证后：

```
{ 'token' : '9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b'  }
```

请注意：默认情况下 `obtain_auth_token` 视图显式地使用JSON请求和响应，会忽略你在settings中关于渲染器和解析器的配置。

默认情况下，没有权限或限流机制应用于 `obtain_auth_token` 视图。如果要使用限流机制，则需要重写视图类，在其中添加 `throttle_classes` 属性。

如果你需要`obtain_auth_token`视图的自定义版本，可以继承`ObtainAuthToken`视图类，并在URL配置中使用新的子类来实现。

例如，你可以返回 `token` 值之外的其他用户信息：

```
from rest_framework.authtoken.views import ObtainAuthToken
from rest_framework.authtoken.models import Token
from rest framework.response import Response

class CustomAuthToken(ObtainAuthToken):

    def post(self, request, *args, **kwargs):
        serializer = self.serializer_class(data=request.data,
                                           context={'request': request})
        serializer.is_valid(raise_exception=True)
        user = serializer.validated_data['user']
        token, created = Token.objects.get_or_create(user=user)
        return Response({
            'token': token.key,
            'user_id': user.pk,
            'email': user.email
        })
```

要同步修改 `urls.py`：

```
urlpatterns += [
path('api-token-auth/', CustomAuthToken.as_view()) 3
]
```

- **使用Admin后台生成令牌**

也可以使用Django的Admin后台手动生成令牌，如下所示：

`your_app/admin.py`：

```
from rest framework.authtoken.admin import TokenAdmin

TokenAdmin.raw id fields = ('user',)
```

- **使用Django的manage.py命令**

从3.6.4版本开始，可以通过下面的命令为用户生成令牌

```
python manage.py drf_create_token <username>
```

结果如下：

```
Generated token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b for user   user1
```

如果你想为用户重新生成令牌，比如令牌已经不安全了或者泄露了的情况下，可以添加 `-r` 参数：

```
python manage.py drf_create_token -r  <username>
```

SessionAuthentication

这种认证方式，使用了Django默认的会话后端，适合AJAX客户端等运行在同样会话上下文环境中的模式，这也是DRF默认的认证方式之一。

可以使用DRF提供的登录页面测试这一功能。

如果认证成功 `SessionAuthentication` 提供下面的属性

- `request.user` ：设置为一个Django的 `User` 类的实例
- `request.auth` ：设置为None

认证不成功将返回 `HTTP 403 Forbidden` 响应，没有额外的头部信息。

如果你正在使用类似AJAX风格的API，并采用SessionAuthentication认证，当使用不安全的HTTP方法，比如 `PUT` ，`PATCH` ，`POST` 或者 `DELETE` ，你必须确保提供了合法的CSRF 令牌。参考Django的CSRF相关章节。

DRF框架中的CSRF验证与标准Django的CSRF验证的工作方式略有不同，因为需要同时支持同一视图的会话和非会话身份验证。这意味着只有经过身份验证的请求才需要CSRF令牌，匿名请求可以在没有CSRF令牌的情况下发送。此行为不适用于登录视图，登录视图应始终应用CSRF验证。

RemoteUserAuthentication

使用Django的auth框架的认证功能。

首先你必须在你的 `AUTHENTICATION_BACKENDS` 配置中，使用 `django.contrib.auth.backends.RemoteUserBackend` (或 继承

如果认证成功， `RemoteUserAuthentication` 它) 。提供下面的属性：

- `request.user` ：设置为一个Django的 `User` 类的实例
- `request.auth` ：设置为None

三、自定义认证框架

自定义认证框架步骤：

1. 继承`BaseAuthentication`类
2. 重写`.authenticate(self, request)`方法，认证成功时返回一个`(user, auth)`二元元组，否则返回`None`。当然，某些情况下，你可能需要弹出一个`AuthenticationFailed`异常。

建议的认证机制：

- 如果未尝试身份验证，则返回None。继续进行任何其他正在使用的身份验证方案。
- 如果尝试身份验证但失败了，则引发`AuthenticationFailed`异常。立即返回错误响应，无论是否进行任何权限检查，也不继续进行任何其他身份验证方案。（这条可以讨论一下）

当认证失败时，你也可以重写`.authenticate_header(self, request)`方法，为`WWW-Authenticate`头部属性添加`HTTP 401 Unauthorized`响应。否则默认进行`HTTP 403 Forbidden`响应。

自定义认证类后，也可以将它作为配置参数，进行全局配置，只是在引用路径的时候，需要注意一下：

比如我们在某个app下创建一个auth模块，再写入MyAuthentication类：

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES':
(   ...
        'app.auth.MyAuthentication',
    )
}
```

下面的例子，对在请求头部中，将用户名保存在 'X_USERNAME'属性中的请求，进行认证：

```
from django.contrib.auth.models import User
from rest_framework import  authentication
from rest framework import exceptions

class MyAuthentication(authentication.BaseAuthentication):
    def authenticate(self, request):
        username = request.META.get('X_USERNAME') # 获取用户名信息
        if not username:
            return None
```

```
10
11          try:
12              user = User.objects.get(username=username) # 查找用户
13          except User.DoesNotExist:
14              raise exceptions.AuthenticationFailed('No such user') #如果用户
    不存在
15
16          return (user, None)
```

四、第三方模块

下面是一些可用的第三方认证模块：

- Django OAuth Toolkit

支持 OAuth 2.0 ，支持Python 2.7 和 Python 3.3+，文档很好，推荐使用。

```
1   pip install django-oauth-toolkit
```

需要做下面的配置：

```
1   INSTALLED_APPS = (
2       ...
3       'oauth2_provider',
4   )
5
6   REST_FRAMEWORK =
7       { 'DEFAULT_AUTHENTICATION_CLASSES':  (
8           'oauth2_provider.contrib.rest_framework.OAuth2Authentication',
9       )
10  }
```

- Django REST framework OAuth： `pip install djangorestframework-oauth`
- JSON Web Token Authentication
- Hawk HTTP Authentication
- HTTP Signature Authentication
- Djoser
- django-rest-auth
- django-rest-framework-social-oauth2
-

django-rest-knox
- drfpasswordless

一、权限Permissions

DRF的权限类位于permissions模块中。

连同认证和限流，权限决定是否应该接收请求或拒绝访问。

权限检查始终在视图的最开始处执行，在继续执行任何其他代码之前。 权限检查通常会使用 `request.user` 和 `request.auth` 属性中的身份认证信息来决定是否允许传入请求。

在不同类别的用户访问不同类别API的过程中，使用权限来控制访问的许可。

最简单粗暴的权限设置：允许任何经过身份验证的用户访问API，并拒绝任何未经身份验证的用户。这对应于DRF框架中的 `IsAuthenticated` 类。

稍微严格点的权限控制是对经过身份验证的用户的允许完全访问（可读可写），但对未经身份验证的用户只允许读取类型访问。这对应于DRF框架中的 `IsAuthenticatedOrReadOnly` 类。

如何确定权限

DRF框架中的权限始终被定义为一个权限类的列表，表示拥有列表中所有类型的权限。

在运行视图的主体代码之前，系统会检查列表中的每个权限。 如果任何权限检查失败，将会抛出一个 `exceptions.PermissionDenied` 或 `exceptions.NotAuthenticated` 异常，并且视图的主体代码将不会运行。

当权限检查失败时，将返回 `"403 Forbidden"` 或 `"401 Unauthorized"` 响应，具体根据以下规则：

- 请求已成功通过身份验证，但不具备访问权限。 — 返回403 Forbidden响
- 应。请求未通过身份认证，并且最高优先级的认证类未使用 `WWW-Authenticate` 403 Forbidden响应。
- 请求未通过身份认证，但是最高优先级的认证类使用了 `WWW-Authenticate` HTTP 401未经授权的响应，并附带适当的WWW-Authenticate报头。

`WWW-`

`Authenticate`*将返回带有适当标头的HTTP 401未经授权的响应。*

设置对象级别的权限

DRF框架还支持对象级别的权限设置。对象级权限用于确定是否允许用户操作特定的对象（通常是模型实例）。

当调用 `.get_object()` 方法时，由DRF框架的通用视图运行对象级权限检测。 与视图级别权限一样，如果不允许用户操作给定对象，则会抛出 `exceptions.PermissionDenied` 异常。

如果你正在编写自己的视图并希望强制执行对象级权限检测，或者你想在通用视图中重写 `get_object` 方法，那么你需要在检索对象的时候显式地调用视图上的 `.check_object_permissions(request, obj)` 方法。这将抛出 `PermissionDenied` 或 `NotAuthenticated` 异常；或者如果视图具有适当的权限，则返回。

例如：

```
def get_object(self):
    obj = get_object_or_404(self.get_queryset(), pk=self.kwargs["pk"])
    self.check_object_permissions(self.request, obj)
    return obj
```

注意：出于性能考虑，通用视图在返回对象列表时不会自动将对象级权限应用于查询集中的每个实例上面。通常，当你使用对象级权限时，你还需要适当地过滤查询集，以确保用户只能看到他们被允许查看的实例。

下面是两个方法的源码：

```
    def check_permissions(self, request):
        """
        Check if the request should be permitted.
        Raises an appropriate exception if the request is not permitted.
        """
        for permission in self.get_permissions():
            if not permission.has_permission(request, self):
                self.permission_denied(
                request, message=getattr(permission, 'message', None) 10
                )

    def check_object_permissions(self, request, obj):
        """
        Check if the request should be permitted for a given  object.
        Raises an appropriate exception if the request is not permitted.
        """
        for permission in  self.get_permissions():
            if not permission.has_object_permission(request, self,  obj):
                self.permission_denied(
                request, message=getattr(permission, 'message', None) 21
```

)

设置权限策略

默认权限策略可以使用 `DEFAULT_PERMISSION_CLASSES` 配置项进行全局设置。比如：

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES':  (
    'rest_framework.permissions.IsAuthenticated', 4
    )
}
```

如果未指定，则此设置默认为允许无限制的访问，也就是下面的配置：

```
'DEFAULT_PERMISSION_CLASSES': (
'rest_framework.permissions.AllowAny', 3
)
```

你还可以为使用基于 `APIView` 的类视图在每个视图或每个视图集的上设置权限策略。

```
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response
from rest_framework.views import APIView

class ExampleView(APIView):
    permission classes = (IsAuthenticated,)    # 看这里！注意参数形式

    def get(self, request, format=None):
        content = {
            'status': 'request was  permitted'
        }
        return Response(content)
```

或者为使用 `@api_view` 装饰器的基于函数的视图设置权限。

```python
from rest_framework.decorators import api_view, permission_classes
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response

@api_view(['GET'])
@permission_classes((IsAuthenticated, ))   # 看这里！注意参数形式
def example_view(request, format=None):
    content = {
        'status': 'request was permitted'
    }
    return Response(content)
```

注意： 当你通过类属性或装饰器设置新的权限类时，会忽略settings中设置的默认权限列表。也就是粒度越细的权限级别越高，这也符合我们通常的逻辑思维。

DRF还提供一种权限的关系运算操作，可以实现组合类权限，也就是与或非运算。除了花哨，没啥特别的，参考下面的例子：

```python
from rest_framework.permissions import BasePermission, IsAuthenticated, SAFE_METHODS
from rest_framework.response import Response
from rest_framework.views import APIView

class ReadOnly(BasePermission):
    def has_permission(self, request, view):
        return request.method in  SAFE_METHODS

class ExampleView(APIView):
    permission_classes = (IsAuthenticated|ReadOnly,) # 看这里，注意竖线表示‘或者’

    def get(self, request, format=None):
        content = {
        'status': 'request was permitted' 15
        }
        return Response(content)
```

二、API参考

AllowAny

`AllowAny` 权限类允许不受限制的访问，并且**不管该请求是否通过身份验证或未经身份验证。**

一般来说，我们不用专门指定此权限，因为你可以通过使用空列表或元组进行权限设置来获得相同的结果，但显式地指定可以使我们地意图更明确。

IsAuthenticated

`IsAuthenticated` 权限类将拒绝任何未经身份验证的用户的访问，通过身份验证的用户则拥有所有权限。 如果你希望你的API仅供注册用户访问，则此权限比较合适。

IsAdminUser

除非 `user.is_staff` 属性的值 `True` ，否则 `IsAdminUser` 权限类将拒绝来者访问API。为

`user.is_staff` 属性是Django的auth框架中，管理员的类别，可以在Admin后台中查看和指定，也可以通过createsuperuser命令创建。如果你希望你的API只能被部分受信任的管理员访问，则此权限比较适合。

IsAuthenticatedOrReadOnly

`IsAuthenticatedOrReadOnly` 权限类允许经过身份验证的用户执行任何请求。未经授权的用户请求，只有当请求方法是“安全”的（即 `GET` ， `HEAD` 或 `OPTIONS` 之一时），才允许访问。

如果你希望你的API允许匿名用户读取，并且只允许对通过身份验证的用户可读写，则此权限比较适合。这也是大多数公开服务的选择。

DjangoModelPermissions

Django模型级别的权限。

此权限类与Django标准的 `django.contrib.auth` model权限相关。此权限只能应用于具有 `.queryset` 属性集的视图。只有在用户通过身份验证并分配了相关模型权限的情况下，才会被授予此权限。

- `POST` 请求要求用户对模型具有 `add` 权限
- `PUT` 和 `PATCH` 要求用户对模型具有 `change`

权限。

- 请求要求用户对模型具有 权限。

- `DELETE` 请求想要求用户对模型具有 `delete` 权限。

也可以通过自定义模型权限，重写以上的默认行为。例如，你可能希望为 `GET` 请求包含一个 `view` 模型的权限。

要使用自定义模型权限，请覆盖 `DjangoModelPermissions` 并设置 `.perms_map` 属性。

如果你在重写了 `get_queryset()` 方法的视图中使用此权限，有可能这个视图上却没有 `queryset` 属性。在这种情况下，建议使用保护性的查询集来标记视图，以便确定所需的权限。比如：

```
1   queryset = User.objects.none()  # Required for DjangoModelPermissions
```

DjangoModelPermissionsOrAnonReadOnly

类似 `DjangoModelPermissions` ，但允许未经身份验证的用户对API的只读权限。

DjangoObjectPermissions

Django的模型的对象级别的权限！粒度最细！

此权限类与Django标准的对象权限框架相关联，该框架允许为模型上的每个对象设置权限。为了使用此权限类，你还需要添加支持对象级权限的权限后端，例如django-guardian。

与 `DjangoModelPermissions` 一样，此权限只能应用于具有 `.queryset` 属性或 `.get_queryset()` 方法的视图。只有在用户通过身份验证并且具有相关的每个对象权限 和相关的模型权限 后，才会被授予此权限。

- `POST` 请求要求用户对模型实例具有 `add` 权限。
- `PUT` 和 `PATCH` 请求要求用户对模型示例具有 `change` 权限。
- `DELETE` 请求要求用户对模型示例具有 `delete` 权限。

与 `DjangoModelPermissions` 一样，你可以通过重写 `DjangoObjectPermissions` 并设置 `.perms_map` 属性来使用自定义模型权限。

三、自定义权限

很显然DRF自带的权限类别太简单了，可能我们需要大量的自定义权限类别。

要自定义权限，需要继承 `BasePermission` 类，并实现以下方法中的一个或两个：

- `.has_permission(self, request, view)`
- `.has_object_permission(self, request, view, obj)`

如果请求被授予访问权限，方法应该返回 `True` ，否则返回 `False` 。

如果你需要测试请求是读取操作还是写入操作，则应该根据permissions模块中常量 `SAFE_METHODS` 的值检查请求方法， `SAFE_METHODS` 是包含 `'GET'` ， `'OPTIONS'` 和 `'HEAD'` 的元组。例如：

```
if request.method in permissions.SAFE_METHODS:
    # 检查只读请求的权限
else:
    # 检查写入请求的权限
```

注意：仅当视图级的 `has_permission` 方法检查通过时，才会调用实例级的 `has_object_permission` 方法。另外，**为了运行实例级别检查，视图代码应显式地调用 `.check_object_permissions(request, obj)` 方法。如果你使用的是通用视图，那么默认会自动为你处理。**

如果权限测试失败，自定义地权限类将引发 `PermissionDenied` 异常。要更改与异常关联的错误信息，请直接在自定义地权限类中添加一个 `message` 属性。否则将使用 `PermissionDenied` 的 `default_detail` 属性。

```
from rest_framework import  permissions

class CustomerAccessPermission(permissions.BasePermission):
    message = 'Adding customers not allowed.' # 自定义异常提示信息

    def has_permission(self, request, view):
        ...
```

下面是一个自定义权限类的例子，根据黑名单检查传入请求的IP地址，如果该IP在黑名单中，则拒绝请求。

```python
from rest_framework import  permissions

class BlacklistPermission(permissions.BasePermission): # 继 承
    """
    全局的黑名单ip检查
    """

    def has_permission(self, request, view):
        ip_addr = request.META['REMOTE_ADDR'] # 获取请求方的ip
        blacklisted = Blacklist.objects.filter(ip_addr=ip_addr).exists() # ORM过滤查询
        return not blacklisted # 返回一个布尔值
```

除了针对所有传入请求运行的全局权限设置外，还可以创建对象级权限，这些权限仅影响特定对象实例的操作行为。例如：

```python
class IsOwnerOrReadOnly(permissions.BasePermission):
    """
    Object-level permission to only allow owners of an object to edit it.
    Assumes the model instance has an `owner` attribute.
    """

    def has_object_permission(self, request, view, obj):
        # Read permissions are allowed to any  request,
        # so we'll always allow GET, HEAD or OPTIONS requests.
        if request.method in permissions.SAFE_METHODS:
            return True

        # Instance must have an attribute named `owner`.
        return obj.owner == request.user
```

请注意，通用视图将检查适当的对象级权限，但如果你正在编写自己的自定义视图，则需要确保检查自己的对象级权限检查，也就是说这活得你自己干。你可以通过在获得对象实例后从视图中调用 `self.check_object_permission(request, obj)` 来执行此操作。如果任何对象级权限检查失败，此调用将引发对应的 `APIException` ，否则将简单地返回。

四、第三方模块

下面是和DRF权限功能相关的一些第三方模块

- Composed Permissions

- REST Condition
- DRY Rest Permissions
- Django Rest Framework Roles
- Django REST Framework API Key
- Django Rest Framework Role
- Filters django-guardian

缓存Caching

Django为类视图提供了一个 `method_decorator` 装饰器，用于为类视图添加缓存类别的装饰器，比如 `cache_page` 和 `vary_on_cookie`。

```
from rest_framework.response import Response
from rest_framework.views import APIView
from rest_framework import viewsets
from django.utils.decorators import method_decorator
from django.views.decorators.cache import  cache_page
from django.views.decorators.vary import  vary_on_cookie


class UserViewSet(viewsets.Viewset):

    # 为每个用户缓存2个小时的请求url
    @method_decorator(cache_page(60*60*2))
    @method_decorator(vary_on_cookie)
    def list(self, request, format=None):
        content = {
            'user_feed':  request.user.get_user_feed()
        }
        return Response(content)

class PostView(APIView):

    # Cache page for the requested url
    @method_decorator(cache_page(60*60*2))
    def get(self, request, format=None):
        content = {
            'title': 'Post title',
            'body': 'Post content'
        }
        return Response(content)
```

注：该`cache_page`装饰只缓存 `GET`，并`HEAD`与状态200的响应。

一、限流Throttling

限流类似于权限机制，因为它也决定是否接受当前请求。限流可以形象地比喻为节流阀，指示一种临时状态，用于控制客户端在某段时间内允许向API发出请求的次数，也就是频率。

你的API可能对未经身份验证的请求具有较强的访问次数限制，而对经身份验证的请求一般不怎么限制访问次数。

另一个你可能希望使用多个节流阀的场景是，如果您需要对API的不同部分施加不同的约束，因为某些服务需要特别的资源，防止服务器负荷过重，或者IO读取等待时间过长等等。

也可以同时使用多种限流措施，以同时应用突发节流率和持续节流率。例如，你可能希望将用户限制为每分钟最多60个请求，每天1000个请求。

限流不一定单指访问次数的限制，也可以是别的形式。例如，存储服务可能还需要限制带宽，而付费数据服务可能需要限制正在访问的特定记录的数量。

与权限和认证机制一样，可以同时使用多个节流阀，也是以类的列表的方式。在运行视图的主体代码之前，会逐个检查列表中的限流措施。如果任何限流检查失败，将引发`Exceptions.Throttled`异常，并且视图主体将不运行。

DRF默认的限流是使用Django的缓存机制，它的数据结构如下：

```
cache =  {

'注册用户的id或者匿名用户的ip地址': [timestamp,timestamp,timestamp,timestamp],
'192.168.1.112':
[...,2019.10.1.12:00:11,2019.10.1.12:00:08,2019.10.1.12:00:03,],
'jack':[...,2019.10.1.12:00:10,2019.10.1.12:00:02,2019.10.1.12:00:01,],
...

}
```

注意，时间戳在列表里是有顺序的，由最近到最早。

阅读该模块的源码会对你理解限流非常有帮助。

配置限流机制

在settings中，通过`DEFAULT_THROTTLE_CLASSES`和`DEFAULT_THROTTLE_RATES`为限流机制

设置全局性的配置 ， 例如：

配置项，

```
REST_FRAMEWORK =
    { 'DEFAULT_THROTTLE_CLASSES':  (
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle'
    ),
    'DEFAULT_THROTTLE_RATES': {
        'anon': '100/day',   # 未认证用户一天只许访问100次
        'user': '1000/day'   # 认证用户一天可以访问1000次
    }
}
```

`DEFAULT_THROTTLE_RATES` 配置项中的时间周期单位可以使用 `second`，`minute`，`hour` 或 者 `day` 。

应用限流机制

也可以为使用APIView，基于类的视图添加视图级别的限流措施：

```
from rest_framework.response import Response
from rest_framework.throttling import UserRateThrottle    #导 入
from rest_framework.views import APIView

class ExampleView(APIView):
    throttle classes =  (UserRateThrottle,)   # 看这里！注意参数形式

    def get(self, request, format=None):
        content = {
            'status': 'request was  permitted'
        }
        return Response(content)
```

对于api_view装饰器装饰的基于函数的视图，也是一样的：

```
@api_view(['GET'])
@throttle_classes([UserRateThrottle])    # 看这里，注意参数形式
def example_view(request,  format=None):
    content = {
        'status': 'request was  permitted'
    }
    return Response(content)
```

如何识别客户端?

既然要限流，那么必须识别客户端！那么DRF是如何判断和区分当前客户端的身份的呢?

DRF利用HTTP报头的 `'x-forwarded-for'` 或WSGI中的 `'remote-addr'` 变量来唯一标识客户端的IP地址。如果存在 `'x-forwarded-for'` 头部属性，则使用它，否则将使用WSGI中 `'remote-addr'` 变量的值。

在代理的情况下，如果想严格标识唯一的客户端IP地址，需要首先设置 `NUM_PROXIES` 来配置API后面运行的应用程序代理的数量。此设置应为大于等于0的整数。如果设置为非零，则一旦排除了任何应用程序代理IP地址，客户端IP将被标识为 `'x-forwarded-for'` 头中的最后一个IP地址。如果设置为零，则 `'remote-addr'` 的值将始终用作标识IP地址。重要的是要清楚，如果配置了 `NUM_PROXIES` ，那么NAT（网络地址转换）网关后面的所有客户机都将被视为单个客户机。

设置缓存

我们还可以在DRF中为限流措施设置缓存功能。

DRF框架提供的限流类需要使用Django的缓存后端。应该确保已经设置了适当的缓存设置。默认的 `LocMemCache` 缓存后端就比较合适了。

如果需要使用 `default` 以外的缓存，可以通过自定义Throttle类并设置 `cache` 属性来实现。例如:

```
from django.core.cache import  caches

class  CustomAnonRateThrottle(AnonRateThrottle):
    cache = caches['alternate']
```

自定义限流类后，不要忘了在settings中进行配置，或者在视图中添加类属性。

二、API 参考

DRF的限流类都定义在throttling模块中，这个模块很简单，一共250多行代码而已，有兴趣可以读读源代码。

主要包括下面几个类:

- BaseThrottle：限流类的基类，用于占位，提供三个方法 `allow_request` 、 `get_ident` 和 `wait`

- SimpleRateThrottle：继承了BaseThrottle，添加和重写了一些方法，重点是添加了`get_cache_key`方法，但你必须自己实现该方法。

- AnonRateThrottle：继承了SimpleRateThrottle，仅仅是重写了 `get_cache_key` 方法
- UserRateThrottle：继承了SimpleRateThrottle，仅仅是重写了 `get_cache_key` 方法
- ScopedRateThrottle：继承了SimpleRateThrottle，重写了 `get_cache_key` 和 `allow_request` 方法

我们一般使用后三个类。

AnonRateThrottle

`AnonRateThrottle` 只会限制未经身份验证的用户。传入的请求的IP地址用于生成一个唯一的密钥。

允许的请求频率由以下各项之一确定（按优先顺序）：

- 类 的`rate` 属性，可以通过继承 `AnonRateThrottle` 并设置该属性来修改这个值。优先级高。
- settings配置文件中 `DEFAULT_THROTTLE_RATES['anon']` 配置项的值。优先级低。

`anonratetrottle` 适用于想限制来自未知用户的请求频率的情况。

UserRateThrottle

`UserRateThrottle` 用于限制已认证的用户在整个API中的请求频率。用户ID用于生成唯一的密钥。未经身份验证的请求将使用传入的请求的IP地址生成一个唯一的密钥。

允许的请求频率由以下各项之一确定（按优先顺序）：

- 类 的`rate` 属性，可以通过继承 `UserRateThrottle` 并设置该属性来修改这个值。优先级高。
- settings配置文件中 `DEFAULT_THROTTLE_RATES['user']` 配置项的值。优先级低。

一个API可能同时会进行多个 `UserRateThrottles` 类型的限流措施，比如每小时限制100次的同时每天限制1000次。那么在这种情况下，就需要自定义限流类。继承 `UserRateThrottle` 类，然后为每个子类添加一个 `scope` 类属性，如下所示：

```
1  class BurstRateThrottle(UserRateThrottle): # 第一个限流子类
2      scope = 'burst'      # 主要用于控制爆发期的访问频率
3
4  class  SustainedRateThrottle(UserRateThrottle):#  第二个限流子类
5      scope = 'sustained' # 主要用于控制持续时期的访问频率
```

定义后，还需要在settings中配置：

```
REST_FRAMEWORK =
    { 'DEFAULT_THROTTLE_CLASSES': (
        'example.throttles.BurstRateThrottle',     # 配置我们自定义的限流类
        'example.throttles.SustainedRateThrottle'
    ),
    'DEFAULT_THROTTLE_RATES': {
        'burst': '60/min',   # 每分钟最多60次，控制爆发期
        'sustained': '1000/day'   #  每天最多100次，控制持续访问
    }
}
```

`UserRateThrottle` 适用于限制每个用户的全局访问频率。

ScopedRateThrottle

`ScopedRateThrottle` 类用于限制对APIs特定部分的访问，也就是视图级别的限流，不是全局性的。只有当正在访问的视图包含 `throttle_scope` 属性时，才会应用此限制。然后，通过将视图的“scope”属性值与唯一的用户ID或IP地址连接，生成唯一的密钥。

允许的请求频率由 `scope` 属性的值在 `DEFAULT_THROTTLE_RATES` 中的设置确定。

看下面的例子：

```
class ContactListView(APIView):
    throttle_scope = 'contacts'   # 定义scope属性，并提供一个字符串，用于去settings中查找设置的值
    ...

class ContactDetailView(APIView):
    throttle_scope = 'contacts' # 同 上
    ...

class UploadView(APIView):
    throttle_scope = 'uploads' # 同 上
    ...
```

不要忘了添加下面的配置：

```
REST_FRAMEWORK =
    { 'DEFAULT_THROTTLE_CLASSES': (
        'rest_framework.throttling.ScopedRateThrottle',    # 注册限流类
    ),
    'DEFAULT_THROTTLE_RATES': {
        'contacts': '1000/day',   # 字典的键，对应视图中的scope属性
        'uploads': '20/day'
    }
}
```

三、自定义限流类

要自定义限流类，请参照下面的步骤：

1. 继承 `BaseThrottle`类
2. 实现 `allow_request(self, request, view)` 方法。如果请求应该被允许，那么方法应该返回 `True`，否则返回 `False` 。
3. 可额外实现 `wait()` 方法。该方法应该返回一个建议的等待秒数（只有等待相应的秒数后，用户才可以尝试下一个请求），或返回None。只有当 `allow_Request ()` 方法返回 `False` 时，才会调用 `wait ()` 方法。

如果实现了 `wait()` 方法，并且当前请求被限流了，那么在响应头部将包含一个 `Retry-After` 属性。

下面是一个自定义限流类的例子，它将随机限制每10个请求中的一个：

```
import random

class RandomRateThrottle(throttling.BaseThrottle):
    def allow_request(self, request, view):
        return random.randint(1, 10) != 1
```

一、过滤Filtering

DRF的过滤器类都定义在filters模块中，这个模块也很简单，300多行代码而已，主要定义了下面四个类：

- BaseFilterBackend：基类，被继承。
- SearchFilter：继承BaseFilterBackend类
- OrderingFilter：继承BaseFilterBackend类
- DjangoObjectPermissionsFilter：继承BaseFilterBackend类。3.9版本后废除。

DRF通用列表视图的默认行为是返回一个模型的全部queryset，比如说你有一个用户的模型，当前存了1万条用户信息，那么默认会将这1万条记录全部取出。而通常情况下，我们是不想也不需要一次性全部取出来的，只需要其中的一部分就行。这就需要我们对查询的结果进行过滤。

最简单的过滤方法就是在任意继承了 `GenericAPIView` 的视图中重写 `.get_queryset()` 方法，使用这个方法来过滤查询结果。

根据用户进行过滤

你可能想要过滤queryset，只返回与发出请求的当前已验证的用户相关的结果。这可以使用 `request.user` 的值进行过滤来实现。

例如：

```
from myapp.models import Purchase
from myapp.serializers import PurchaseSerializer
from rest_framework import generics

class  PurchaseList(generics.ListAPIView):
    serializer_class  = PurchaseSerializer

    def get_queryset(self):   #1. 重写该方法
        """
        这个视图将返回一个列表，只显示当前用户的购物清单，而不是所有用户的购物清单
        """
        user = self.request.user # 2.获取用户
        return  Purchase.objects.filter(purchaser=user)   #  3.返回用户的购物清单
```

根据URL进行过滤

还有一种过滤方式，是根据请求的URL的部分内容来过滤查询集。

例如，对于下面形式的url:

```
path('purchases/<str:username>/', PurchaseList.as_view()),
```

你就可以写一个视图，返回基于URL中的username参数进行过滤的结果。

```
class  PurchaseList(generics.ListAPIView):
    serializer_class = PurchaseSerializer

    def get_queryset(self):
        """
        This view should return a list of all the purchases for
        the user as determined by the username portion of the URL.
        """
        username = self.kwargs['username'] # 与上面的例子，不同之处只是获取username的方式变了
        return Purchase.objects.filter(purchaser__username=username)
```

根据查询参数进行过滤

实际上我们还可以通过URL中携带的参数来过滤查询集。也就是上一节更广义的形式。

比如我们可以通过重写 `.get_queryset()` 方法来处理像 `http://example.com/api/purchases?username=denvercoder9` 这样的URL，并且只有在URL中包含 `username` 参数时，才过滤查询集：

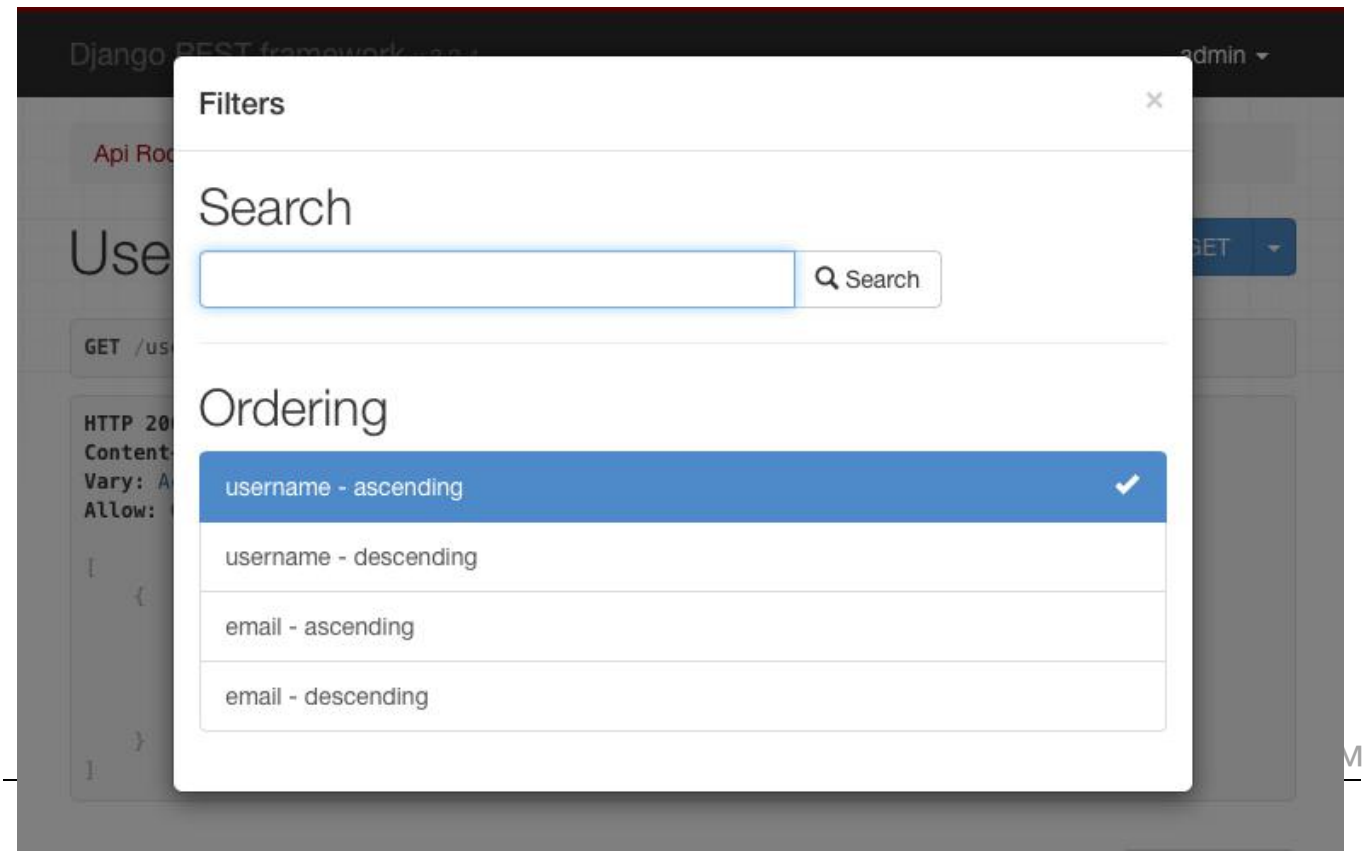
```python
class  PurchaseList(generics.ListAPIView):
    serializer_class = PurchaseSerializer

    def get_queryset(self):
        """
        Optionally restricts the returned purchases to a given user,
        by filtering against a `username` query parameter in the URL.
        """
        queryset = Purchase.objects.all() # 这是保底的查询集
        username = self.request.query_params.get('username', None) # 注意获取参数的方法
        if username is not None:
            queryset = queryset.filter(purchaser_username=username) # 这是过滤后的查询集
        return queryset # 总是能返回一个查询集
```

二、通用过滤

除了能够重写默认的查询集，DRF框架还支持通用的过滤后端，让你可以轻松地构建复杂的检索器和过滤器。

通用过滤器也可以在可浏览的API和Admin后台API中显示为HTML搜索或过滤控件。



默认的过滤器后端可以在全局设置中使用 `DEFAULT_FILTER_BACKENDS` 来配置。例如。

```
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS':
('django_filters.rest_framework.DjangoFilterBackend',)
}
```

注意了，这个过滤器后端不是DRF自带的，而是django-filter模块提供的。所以需要pip安装。需要注意参数的引用方式，不要犯经验主义错误。

还可以使用基于 `GenericAPIView` 的类视图在每个view或每个viewset基础上设置过滤器后端。

```
import django_filters.rest_framework
from django.contrib.auth.models import User
from myapp.serializers import UserSerializer
from rest_framework import generics

class UserListView(generics.ListAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    filter_backends = (django_filters.rest_framework.DjangoFilterBackend,) # 看这里
```

注意下面的例子，查找一个ID为 `4675` 的产品。以下URL将返回相应的对象或者返回404，具体取决于给定的产品实例是否满足筛选条件。也就是说，你必须同时满足id存在，过滤条件也符合，才能成功或者这个唯一对象。

```
http://example.com/api/products/4675/?category=clothing&max_price=10.00
```

我们还可以同时重写 `.get_queryset()` 方法并使用通用过滤器，并且一切都会按照预期生效。 例如，如果 `Product` 模型与 `User` 模型具有多对多关系，也就是购物清单 `purchase` ，则可能需要编写如下所示的视图：

```
class PurchasedProductsList(generics.ListAPIView):
    """
    Return a list of all the products that the authenticated
    user has ever purchased, with optional filtering.
    """
    model = Product
    serializer_class = ProductSerializer
    filterset_class = ProductFilter

    def get_queryset(self):
        user = self.request.user
        return user.purchase_set.all()
```

三、API 参考

DjangoFilterBackend

`django-filter` 模块包含一个 `DjangoFilterBackend` 类，为DRF提供高度可定制的字段过滤功能。

要使用 `DjangoFilterBackend` ，首先需要安装 `django-filter` 模块。，然后将 `django_filters` 添加到Django的 `INSTALLED_APPS` 列表中。

```
pip install django-filter

//如果同时安装下面的库，可以让过滤的输入表单更美观
pip install django-crispy-forms
```

安装完成后，可以进行下面的配置，以进行全局性的过滤操作：

```
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS':
('django_filters.rest_framework.DjangoFilterBackend',)
}
```

或者将它们添加到单个视图中，进行视图级别的过滤操作：

```
from django_filters.rest_framework import DjangoFilterBackend # 导入！

class UserListView(generics.ListAPIView):
    ...
    filter_backends = (DjangoFilterBackend,) # 看这
```

如果你只需要一个简单的过滤功能，那么也只需要简单地在视图或视图集上添加一个 `filterset_fields` 属性，属性的值是一个元组，列出要过滤的字段，如下所示：

```
class  ProductList(generics.ListAPIView):
    queryset =  Product.objects.all()
    serializer_class = ProductSerializer
    filter_backends = (DjangoFilterBackend,) # 先指定后端
    filterset_fields = ('category', 'in_stock') # 看这里
```

这将为指定的字段自动创建一个 `FilterSet` 类，然后你就可以发送类似下面的url请求了：

```
http://example.com/api/products?category=clothing&in_stock=True
```

Django-filter**模块的默认模式是完全匹配模式**，如果需要自定义匹配模式请查阅 django-filter documentation。

SearchFilter

`SearchFilter` 类是DRF自带的过滤器，支持基于简单的单个查询参数的搜索，并且基于Django admin的搜索功能。

在使用时，可浏览的API页面中将包括一个 `SearchFilter` 控件：

仅当视图中设置了 `search_fields` 属性时，才会应用 `SearchFilter` 类。只支持文本类型字段，例如 `CharField` 或 `TextField` 。 `search_fields`

```
from rest_framework import  filters

class UserListView(generics.ListAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    filter_backends = (filters.SearchFilter,)    # DRF自带的过滤器
    search_fields = ('username',  'email')  # 看这里
```

做了上面的工作，你才可以访问下面形式的url（注意参数）：

```
http://example.com/api/users?search=russell
```

你还可以使用双下划线对ForeignKey或ManyToManyField执行关系查询：

```
search_fields = ('username', 'email', 'profile_profession')     # 注意最后一个元素
```

默认情况下，搜索不区分大小写，并使用部分匹配的模式。实际上，可以同时有多个搜索参数，用空格和/或逗号分隔。 如果使用多个搜索参数，则仅当所有提供的模式都匹配时才在列表中返回对象。

可以通过在 `search_fields` 前面添加各种字符来限制搜索行为：

- `^` 以指定内容开始
- `=` 完全匹配
- `@` 全文搜索（目前只支持Django的MySQL后端）
- `$` 正则搜索

例如：

```
search_fields = ('=username',  '=email')  # 指定用户名和邮箱必须完全一致，不能局部一致
```

默认情况下，url中搜索参数的名字为 `'search'` ，这可以通过 `SEARCH_PARAM` 配置项进行自定义，比如改为 `find` ：

```
http://example.com/api/users?find=russell
```

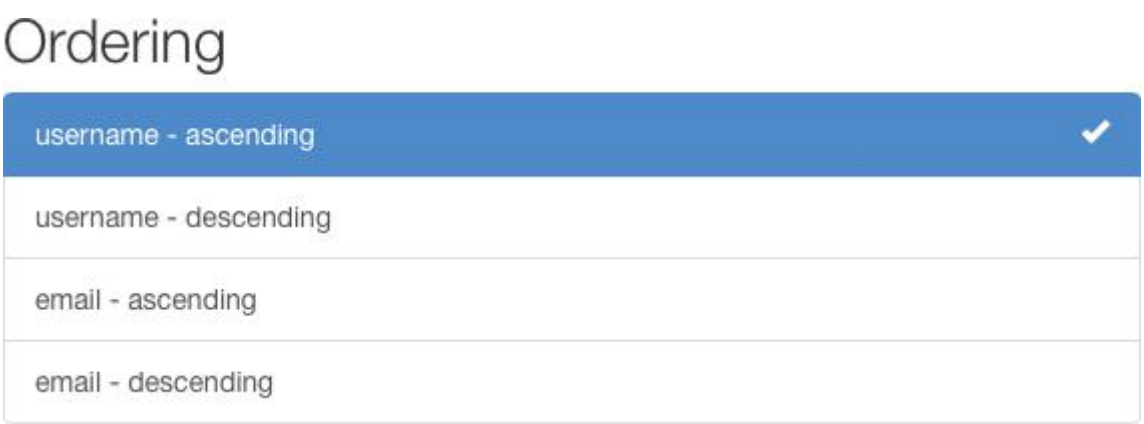
为了根据请求的内容动态地修改查找的字段，可以继承 `SearchFilter` 类，并重写 `get_search_fields()` 方法。例如，下面的搜索类只搜索title字段，如果在请求中包含了 `title_only` 参数。

```
from rest_framework import  filters

class CustomSearchFilter(filters.SearchFilter):
    def get_search_fields(self, view,  request):
        if request.query_params.get('title_only'):
            return ('title',) # 如何条件则只搜索title字段
        return super(CustomSearchFilter, self).get_search_fields(view, request) # 保底的
```

OrderingFilter

`OrderingFilter` 类支持简单的查询参数，以控制查询集的元素顺序。



默认情况下，url中的查询参数名为 `'ordering'` ，可以通过 `ORDERING_PARAM` 配置项进行自定义，和前面的 `search` 一样。

首先看一个根据用户名进行排序的url：

```
http://example.com/api/users?ordering=username
```

客户端还可以为字段名称加上'-'来指定反向排序，如下所示：

```
http://example.com/api/users?ordering=-username
```

也可以指定多个排序：

```
http://example.com/api/users?ordering=account,username
```

指定可以排序的字段

如果不在视图上指定 `ordering_fields` 属性，过滤器默认允许用户对序列化类上的所有可读字段进行排序。建议你明确地指定API可以在哪些字段上进行排序过滤，这有助于防止意外的数据泄漏，例如不小心允许用户针对密码字段或其他敏感数据进行排序。

这一操作可以通过在视图中设置 `ordering_fields` 属性来实现，如下所示：

```
1   class UserListView(generics.ListAPIView):
2       queryset = User.objects.all()
3       serializer_class = UserSerializer
4       filter backends = (filters.OrderingFilter,)
5       ordering fields = ('username',  'email')   # 看这里
```

如果你确定视图正在使用的查询集中不包含任何敏感数据，还可以通过使用特殊值 `'__all__'` 来明确指定可以在所有字段上排序。

```
1   class BookingsListView(generics.ListAPIView):
2       queryset = Booking.objects.all()
3       serializer_class = BookingSerializer
4       filter_backends = (filters.OrderingFilter,)
5       ordering_fields = ' all ' # 看 这 里
```

指定默认的排序字段

如果在视图中设置了 `ordering` 属性，则将把它用作默认的排序。

通常，你可以通过在初始查询语句上设置 `order_by` 参数来控制此操作，但是使用视图中的 `ordering` 参数允许你以某种方式指定排序，然后可以将其作为上下文自动传递到渲染的模板。如果它们用于排序结果的话就能使自动渲染不同的列标题成为可能。（好麻烦…）

```
1   class UserListView(generics.ListAPIView):
2       queryset = User.objects.all()
3       serializer_class = UserSerializer
4       filter_backends =  (filters.OrderingFilter,)
5       ordering_fields = ('username', 'email')
6       ordering =  ('username',)   # 看这里，设置了默认结果按username排序
```

`ordering` 属性可以是一个字符串，或者字符串的列表/元组。

DjangoObjectPermissionsFilter

`DjangoObjectPermissionsFilter`需要和Django Guardian模块一起使用，用于添加自定义的 `view`权限。过滤器将确保查询集只返回用户拥有权限的对象。这个过滤器已经从3.9版本后被废弃，并移动到了 `djangorestframework-guardian` 模块中。

四、自定义通用过滤

要自定义通用过滤后端，需要继承 `BaseFilterBackend` 类，并重写`filter_queryset(self, request, queryset, view)` 方法。该方法应返回一个新的过滤后的查询集。

除了允许客户端执行搜索和过滤之外，通用过滤器后端还可以限制当前请求或用户能够访问的对象。

比如，你可能希望限制用户只能访问他们自己创建的对象：

```
class IsOwnerFilterBackend(filters.BaseFilterBackend):
    """
    Filter that only allows users to see their own objects.
    """
    def filter_queryset(self, request, queryset, view):
        return queryset.filter(owner=request.user)
```

实际上，我们在视图中，通过重写 `get_queryset()` 方法，也能实现上面的操作。但是，编写自定义的过滤器后端，可以让你在不同的视图或所有的API上，方便的重用这一功能。

通用过滤器有时候也需要提供一个可浏览API页面上的过滤器控件。要实现这一功能，你需要实现 `to_html()` 方法，该方法返回过滤器被渲染过的HTML形式。方法的签名如下：

```
to_html(self, request, queryset, view)
```

方法返回的是渲染过的HTML字符串形式。

五、第三方模块

下面是一些第三方模块，提供了一些额外的过滤功能：

- Django REST framework filters package
- Django REST framework full word search filter
- Django URL Filter

- drf-url-filters

一、分页Pagination

DRF框架允许自定制分页样式，也就是说你可以设置每页显示的对象数量。

分页API支持以下操作：

- 将分页链接作为响应内容的一部分。
- 响应头中包含分页链接，如 `Content-Range` 或 `Link` 。

只有在使用通用视图或视图集时才会自动执行分页。如果您使用的是常规APIView，则需要自己调用分页API，以确保响应中包含分页数据。

可以通过将分页类设置为None来关闭分页功能。

设置分页风格

可以使用 `DEFAULT_PAGINATION_CLASS` 和 `PAGE_SIZE` 进行全局性的分页风格设置。例如下面的例子:

```
1   REST_FRAMEWORK = {
2       'DEFAULT_PAGINATION_CLASS':
    'rest_framework.pagination.LimitOffsetPagination',
3   'PAGE_SIZE': 100 4  }
```

注意， `DEFAULT_PAGINATION_CLASS` 和 `PAGE_SIZE` 必须同时设置。**默认情况下，它们都是 `None` ，也就是不分页。**

也可以为单个的视图添加 `pagination_class` 属性，设置视图级别的分页风格。

修改分页风格

一般我们通过继承现有分页类，然后设置类属性的方式来修改分页风格，其实也就是相当于自定义分页类了。

```
class LargeResultsSetPagination(PageNumberPagination): # 继 承
    page_size = 1000    # 记住这几个可以配置的属性的名字
    page_size_query_param = 'page_size'
    max_page_size = 10000

class StandardResultsSetPagination(PageNumberPagination):
    page_size = 100
    page_size_query_param = 'page_size'
    max_page_size = 1000
```

然后你就可以在视图中添加 `pagination_class` 属性，并指定为你刚才自定义的分页类了，如下所示：

```
class BillingRecordsView(generics.ListAPIView):
    queryset = Billing.objects.all()
    serializer_class = BillingRecordsSerializer
    pagination_class = LargeResultsSetPagination # 看这里
```

或者在 `DEFAULT_PAGINATION_CLASS` 中进行全局性配置：

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
'apps.core.pagination.StandardResultsSetPagination'
}
```

二、API参考

在DRF的pagination模块中，定义了四个分页类：

- BasePagination： 基 类
- PageNumberPagination：继承了BasePagination
- LimitOVsetPagination： 继 承 了
- BasePagination CursorPagination：继承了BasePagination

没事可以读一读它的源码，看看别人是怎么写分页功能的。

PageNumberPagination

这种分页风格接收一个url中的页码参数。

请求:

```
GET  https://api.example.org/accounts/?page=4
```

响应:

```
HTTP 200 OK
{
    "count": 1023   #数 量
    "next":  "https://api.example.org/accounts/?page=5",   #上一页url
    "previous": "https://api.example.org/accounts/?page=3", # 下 一 页
    "results": [   # 具体的结果
       …
    ]
}
```

配置

要使用 `PageNumberPagination` 分割，需要进行下面的配置：

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.PageNumberPagination',
'PAGE_SIZE': 100 4 }
```

对于继承了 `GenericAPIView` 的类视图，你也可以通过设 `pagination_class` 属性选择置

`PageNumberPagination` 作为视图级别的分页功能。

属性

`PageNumberPagination` 类包含一系列属性用于设置分页风格：

要启用这些属性，你需要继承 `PageNumberPagination` 类，然后激活你自定义的子类，并添加类属性：

- `django_paginator_class` - 使用Django的分页类。默认是 `django.core.paginator.Paginator` ，它适用于大多数场景。
- `page_size` - 每页包含对象的数量。
- `page_query_param` - 一个字符串，指示查询参数的名字。也就是 `https://api.example.org/accounts/?page=5` 中的那个page字符串。
-

page_size_query_param - 如果设置了，这将是一个字符串值，在url中设置每页对象数量的参数的名字，也就是 https://api.example.org/accounts/?page=5&page_size=100 中的 page_size 。默认为None，表示客户端可能无法控制请求

页面的对象数量。注意，`page_size_query_param`指示的是那个参数的名字，而不是参数的值。

- `max_page_size` - 如果设置了，限制每页最大允许显示的对象数量。只有当`page_size_query_param`设置了的时候才有效。
- `last_page_strings` - 一个字符串的列表或元组。用于表示请求最后一页，默认是`('last',)`，比如`https://api.example.org/accounts/?page=last`
- `template` - 当使用可浏览API功能时，用于渲染分页控制的模板。设置为None，将关闭这一功能。默认是`"rest_framework/pagination/numbers.html"`。

LimitOGsetPagination

这种分页样式类似查找多个数据库记录时使用的语法。客户端使用一个“limit”和一个“oVset”查询参数。limit参数指示要返回的最大对象数，相当于其他样式中的`page_size`属性。oVset参数指示相对于完整的未分页集合的起始偏移位置。也就是相对于完整的查询集，从第一个对象开始oVset往后数指定数目的位置开始，取出limit个对象。

请求:

```
GET https://api.example.org/accounts/?limit=100&offset=400
```

响应:

```
HTTP 200 OK
{
    "count": 1023
    "next":  "https://api.example.org/accounts/?limit=100&offset=500",
    "previous":  "https://api.example.org/accounts/?limit=100&offset=300",
    "results":
[   …
    ]
}
```

配置

要使用`LimitOffsetPagination`风格的分页器，需要如下配置：

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.LimitOffsetPagination'
}
```

你还可以添加一个 `PAGE_SIZE` 配置，相当于设置默认的 `limit` 值，客户端可以不再指定limit。

对于继承了 `GenericAPIView` 的类视图，你也可以通过设 `pagination_class` 属性选择置

`LimitOffsetPagination` 作为视图级别的分页功能。

属性

`LimitOffsetPagination` 类包括下面的属性。要使用这些属性，你首先要继承 `LimitOffsetPagination` 类，并按上面的方式激活子类。

- `default_limit` - 一个数值，表示默认的limit数量。
- `limit_query_param` - url中指示limit的参数名，一个字符串，默认是 `'limit'` 。
- `offset_query_param` - 类似上面，默认为 `'offset'` 。
- `max_limit` - 如果设置了，表示最大允许的limit数量。默认为None。
- `template` - 同前。默认为 `"rest_framework/pagination/numbers.html"` 。

CursorPagination

基于光标的分页显示了一个不透明的“光标”指示器，客户端可以使用它在结果集中分页。这种分页样式只显示正向和反向控件，不允许客户端导航到任意位置。

基于光标的分页要求结果集中有一个唯一的、不变的排序方式。这种排序依据通常是记录的创建时间戳，因为这提供了一种一致的排序方式，以便进行分页。

基于光标的分页比其他方案更复杂，但是，它也有以下好处：

- 提供一致的分页视图。如果使用得当， `CursorPagination` 可以确保客户端在翻页时不会看到同一对象两次，即使在分页过程中其他客户端正在插入新对象。
- 支持超大数据集。对于非常大的数据集，使用基于偏移量的分页样式进行分页可能会变得效率低下或不可用。基于光标的分页方案具有固定的时间属性，并且不会随着数据集的增大而减慢。

一些细节和说明

恰当地使用基于光标的分页需要注意一些细节。你需要考虑使用什么排序方式。默认是按 `"-created"` 排序，也就是创建时间的逆序。这需要一个前提，就是在模型上必须有一个 `created`

时间戳字段，并且将呈现一个“时间线”样式的分页视图，首先显示最近添加的项。您可以通过覆盖 `pagination` 类上 `ordering` 属性，或使用 `OrderingFilter` 过滤器类和的 `CursorPagination` 来修改排序。

要正确使用光标分页应该有一个符合以下条件的字段：

- 应该是一个不变的值，例如时间戳或其他在创建时只设置一次的字段。
- 应该是唯一的，或者几乎是唯一的。毫秒精度的时间戳就是一个很好的例子。光标分页的这种实现使用了一种智能的“位置加偏移”样式，允许它正确地支持排序时不严格唯一的值。
- 应该是可以强制为字符串的不可为空的值。
- 不应该是浮点数。精度误差容易导致不正确的结果。
- 该字段应具有数据库索引。

使用不满足这些约束的字段进行排序也许仍然有效，但是将失去光标分页的一些优点。

配置

要使用 `CursorPagination` ，需要进行如下的全局配置：

```
1   REST_FRAMEWORK = {
2       'DEFAULT_PAGINATION_CLASS':
    'rest_framework.pagination.CursorPagination',
3   'PAGE_SIZE': 100 4  }
```

对于继承了 `GenericAPIView` 的类视图，你也可以通过设 `pagination_class` 属性选择置

`CursorPagination` 作为视图级别的分页功能。

属性

`CursorPagination` 类包括下面的属性。要使用这些属性，你首先要继承 `CursorPagination` 类，并按上面的方式激活子类。

- `page_size` :指示每页显示的数量。
- `cursor_query_param`： url中表示‘cursor’分页的参数名。 默认为 `'cursor'` 。
- `ordering` ： 一个字符串，或者字符串的列表。表示用于排序的字段。默认是 `-created` 。
- `template` ： 同前，默认为 `"rest_framework/pagination/previous_and_next.html"` .

三、自定义分页风格

自定义分页类的步骤：

1. 继承 `pagination.BasePagination`

2. 重写 `paginate_queryset(self, queryset, request, view=None)` 和 `get_paginated_response(self, data)` 方法。

- `paginate_queryset` 方法接收初始的查询集，并返回一个可迭代的对象，这个对象只包含当前请求页面的数据。
- `get_paginated_response` 方法接收分页的数据，返回渲染过的实例。用这个方法替代DRF默认的Response方法，可以在返回的响应内容中，添加一些额外分页相关的数据和链接。

看一个例子，假设我们想用一个修改过的格式替换默认的分页输出样式，该格式在嵌套的“links”键中包含下一个和上一个链接。我们可以这样自定义分页类：

```
class CustomPagination(pagination.PageNumberPagination):
    def get_paginated_response(self, data):
        return Response({
            'links': {
                'next': self.get_next_link(),
            'previous': self.get_previous_link() 7
            },
            'count': self.page.paginator.count,
        'results': data 10
        })
```

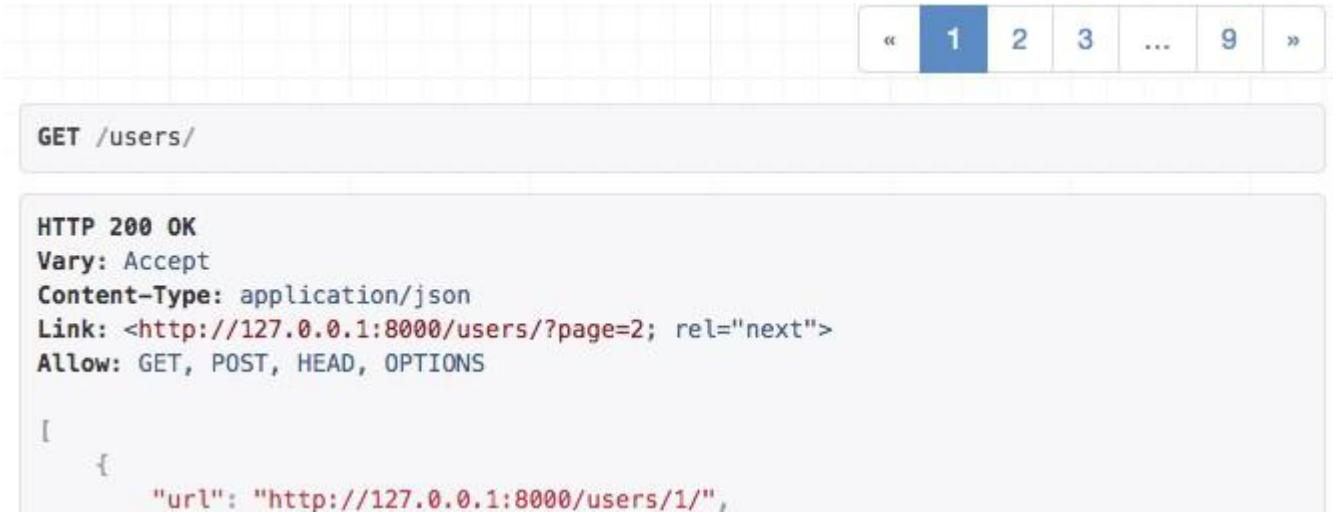
不要忘了配置我们自定义的分页类，否则它是不起作用的：

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
'my_project.apps.core.pagination.CustomPagination',
'PAGE_SIZE': 100 4 }
```

你也可以在DRF框架提供的自动生成模式中使用分页控制，这需要你实现 `get_schema_fields()` 方法。该方法的签名如下：

```
get_schema_fields(self, view)
```

这个方法必须返回 `coreapi.Field` 实例的列表。



四、第三方模块

- DRF-extensions
- drf-proxy-pagination
- link-header-pagination

一、版本控制Versioning

API版本控制允许你在不同的客户端之间更改行为。RESTframework 提供了许多不同的版本控制方案。

版本控制由传入的客户端请求决定，可以基于请求的URL中的版本部分，也可以基于请求头中属性定义的值。

使用DRF进行版本控制

当启用API版本控制后， `request.version` 属性中将包含与当前请求中版本相对应的字符串。

默认情况下，版本控制没有启用， `request.version` 将总是返回 `None` 。

- **根据版本改变行为**

如何改变API的行为取决于你自己，但是一个常见的做法是在新版本中使用不同的序列化样式。例如:

```
1   def get_serializer_class(self):
2       if self.request.version == 'v1':
3           return AccountSerializerVersion1
4       return AccountSerializer
```

- **反向解析URLs中的版本APIs**

REST framework 包含的 `reverse` 函数与版本控制方案相关联。你需要确保将当前 `request` 作为关键字参数传递进去，如下所示。

```
1   from rest_framework.reverse import  reverse
2
3   reverse('bookings-list',  request=request)
```

上述函数将应用任何适用于请求版本的URL转换。例如:

- 如果使用 `NamespacedVersioning`，并且API的版本是'v1'，那么将会查找 `'v1:bookings-list'`，反向解析为类似 `http://example.org/v1/bookings/` 的URL。也就是版本字段形式。

- 如果使用 `QueryParameterVersioning` 并且API的版本是`1.0`，那么返回的URL可能像这样 `http://example.org/bookings/?version=1.0` 。也就是版本参数形
- 式。**版本控制和超链接序列化器**

当将超链接序列化样式与基于URL的版本控制方案一起使用时，请确保将请求作为上下文包含在序列化程序中。

```
1   def get(self, request):
2       queryset = Booking.objects.all()
3       serializer = BookingsSerializer(queryset, many=True, context={'request':
    request})# 看这里
4       return Response({'all_bookings': serializer.data})
```

这样做将会在所有返回的URL中包含适当的版本参数或字段。

配置版本控制

使用 `DEFAULT_VERSIONING_CLASS` 进行版本相关的配置：

```
1   REST_FRAMEWORK = {
2       'DEFAULT_VERSIONING_CLASS':
    'rest_framework.versioning.NamespaceVersioning'
3   }
```

如果没有配置，那么 `DEFAULT_VERSIONING_CLASS` 的默认值为`None`，这种情况时 `request.version` 的值 `None`，也就相当于关闭了版本控制。

还可以在单个视图上设置版本控制方案。但是通常我们不这么做，因为全局使用统一的版本控制方案更有意义。如果你非要这么做，请在视图类中使用 `versioning_class` 属性：

```
1   class ProfileList(APIView):
2       versioning_class = versioning.QueryParameterVersioning
```

其他版本相关设置

以下的配置项也用于版本管理:

- `DEFAULT_VERSION` .:当版本控制信息不存在时用于设置 `request.version` 的默认值，默认 为 `None`。

- `ALLOWED_VERSIONS`:如果设置了此值，将限制提供服务的版本范围，如果客户端请求的版本不在此范围中，则会引发错误。请注意，用于`DEFAULT_VERSION`的值应该总是在`ALLOWED_VERSIONS`设置的范围中（除非是`None`）。该配置默认为`None`，表示不限制。
- `VERSION_PARAM`:url中代表版本相关的参数名，默认值是`'version'`。

你还可以通过自定义版本类，并使用`default_version`，`allowed_versions`和`version_param`三类变量，在每个视图或每个视图集的基础上设置自定义的版本方案。例如，如果你想自定义`URLPathVersioning`的子类：

```
from rest_framework.versioning import URLPathVersioning
from rest_framework.views import APIView

class ExampleVersioning(URLPathVersioning): # 继 承 它
    default_version = ... #指定自己想要的设置
    allowed_versions = ...
    version_param = ...

class ExampleView(APIVIew):
    versioning_class = ExampleVersioning # 在视图中使用
```

二、API 参考

DRF的版本类都位于versioning模块中，这个模块很简单，不到200行代码，主要定义了几个类：

- BaseVersioning：基类，用于继承
- AcceptHeaderVersioning： 继 承 了 BaseVersioning
- URLPathVersioning： 继 承 了 BaseVersioning
- NamespaceVersioning： 继 承 了 BaseVersioning
- HostNameVersioning： 继 承 了 BaseVersioning
- QueryParameterVersioning：继承了BaseVersioning

五个主要类的区别在于，你在哪个地方提供请求的版本信息！是HTTP头部？还是URL的一部分？还是URL的参数？等等

AcceptHeaderVersioning

此版本方案要求客户端将版本要求放在 `Accept` 头部信息中。

下面是一个HTTP请求示例：

```
GET /bookings/ HTTP/1.1
Host: example.com
Accept: application/json; version=1.0 #注意最后的部分
```

在上面的示例请求中，`request.version`属性将返回字符串`'1.0'`。

基于 accept **头部的版本控制通常被认为是最佳实践**，尽管其他版本控制方式可能更适合你的客户端需求。

严格地说`json`媒体类型并不算作包含的参数之一。如果要构建明确指定的公共API，则可以考虑使用vendor 媒体类型。为此，请将渲染器配置为使用自定义媒体类型的基于JSON的渲染器：

```
class BookingsAPIRenderer(JSONRenderer):
    media_type = 'application/vnd.megacorp.bookings+json'
```

这样的话，你的HTTP请求报头会是下面的样子：

```
GET /bookings/ HTTP/1.1
Host: example.com
Accept: application/vnd.megacorp.bookings+json; version=1.0
```

URLPathVersioning

将版本要求作为URL路径的一部分。注意下面的`/v1/`。

```
GET /v1/bookings/ HTTP/1.1
Host: example.com
Accept: application/json
```

你的URL conf中必须包含一个使用`'version'`关键字参数的匹配模式，以便路由器可以获取对应的值，也就是版本信息。

```
urlpatterns = [
    re_path(
        r'^(?P<version>(v1|v2))/bookings/$',
        bookings_list,
    name='bookings-list' 6
    ),
    re_path(
        r'^(?P<version>(v1|v2))/bookings/(?P<pk>[0-9]+)/$',
        bookings_detail,
        name='bookings-detail'
    )
]
```

NamespaceVersioning

对于客户端，此方案与 `URLPathVersioning` 相同。唯一的区别是，它是如何在 Django 应用程序中配置的，因为它使用URL conf中的命名空间而不是URL conf中的关键字参数。

```
GET /v1/something/ HTTP/1.1
Host: example.com
Accept: application/json
```

使用此方案， `request.version` 属性是根据与传入请求的路径匹配的 `namespace` 确定的。

在下面的示例中，我们给一组视图提供了两个可能出现的不同的URL前缀，每个前缀在不同的命名空间下:

```
# bookings/urls.py
urlpatterns = [
    re_path(r'^$', bookings_list, name='bookings-list'),
    re_path(r'^(?P<pk>[0-9]+)/$', bookings_detail, name='bookings-detail')
]

# urls.py
urlpatterns = [
    re_path(r'^v1/bookings/', include('bookings.urls', namespace='v1')),
    re_path(r'^v2/bookings/', include('bookings.urls', namespace='v2'))
]
```

如果你只需要一个简单的版本方案，`URLPathVersioning` 和 `NamespaceVersioning` 都是合适的。`URLPathVersioning` 这种方法可能更适合小型项目，对于更大的项目来说 `NamespaceVersioning` 可能更容易管理。

HostNameVersioning

通过主机名控制版本方案要求客户端将请求的版本指定为URL中主机名的一部分。 例如，以下是对 `http://v1.example.com/bookings/` 的HTTP请求：

```
GET /bookings/ HTTP/1.1
Host: v1.example.com
Accept: application/json
```

默认情况下，此实现期望主机名与下面的正则表达式匹配:

```
^([a-zA-Z0-9]+)\.[a-zA-Z0-9]+\.[a-zA-Z0-9]+$
```

注意，第一组用括号括起来，表示这是主机名的匹配部分。

QueryParameterVersioning

这种版本方案是在 URL 中包含版本信息作为查询参数的方式，例如：

```
GET /something/?version=0.1 HTTP/1.1
Host: example.com
Accept: application/json
```

三、自定义版本方案

要实现自定义的版本方案，只需要继承 `BaseVersioning` 类并重写 `determine_version` 方法。

下面的例子中使用一个自定义的 `X-API-Version` 头部属性来确定所请求的版本。

```
class XAPIVersionScheme(versioning.BaseVersioning):
    def determine_version(self, request, *args, **kwargs):
        return request.META.get('HTTP_X_API_VERSION', None)
```

如果你的版本化方案基于请求的URL，你还需要改变带有版本信息的URL的确定方式。为此，你应该重写类中的 `.reverse()` 方法。有关示例，请参见源代码。

一、内容协商Content negotiation

内容协商：客户端与服务器协商使用哪种数据格式返回给客户端的过程。

DRF框架的内容协商比较简单，根据可用的渲染器、每个渲染器的优先级以及客户端的 `Accept:` 头部属性，确定应将哪种媒体类型返回给客户端。

比如，对于下面的额 `Accept` 属性:

```
1   application/json; indent=4, application/json, application/yaml,  text/html,
    */*
```

那么优先级顺序如下:

1. `application/json;indent=4`
2. `application/json` ,`application/yaml` and `text/html`
3. `*/*`

如果请求的视图只配置了“YAML”和“HTML”的渲染器，那么REST框架将选择在 `renderer_classes` 列表或 `DEFAULT_RENDERER_CLASSES` 设置中中最先列出的渲染器。

二、自定义内容协商

我们一般不建议自定义内容协商的方法，如果非要如此:

1. 继承 `BaseContentNegotiation` 类。
2. 实现 `.select_parser(request, parsers)` 和 `.select_renderer(request, renderers, format_suffix)` 方

`select_parser()` 方法应该返回可用解析器列表中的一个解析器实例。如果没有解析器可以解析情字u，则返回None。

`select_renderer()` 方法应该返回一个二元元组(渲染器实例，媒体类型)，或者弹出一个 `NotAcceptable` 异常。

下面是一个自定义内容协商类，它会选择适当的分析器或渲染器，并忽略客户端的要求。

```
from rest framework.negotiation import BaseContentNegotiation

class IgnoreClientContentNegotiation(BaseContentNegotiation):
```

```
    def select_parser(self, request, parsers):
        """
        Select the first parser in the `.parser_classes` list.
        """
        return parsers[0]

    def select_renderer(self, request, renderers, format_suffix):
        """
        Select the first renderer in the `.renderer_classes` list.
        """
        return (renderers[0], renderers[0].media_type)
```

三、配置内容协商

可以使用 `DEFAULT_CONTENT_NEGOTIATION_CLASS` 进行全局的内容协商配置。比如下面的例子，使用的是我们上面自定义的 `IgnoreClientContentNegotiation` 类：

```
REST_FRAMEWORK = {
    'DEFAULT_CONTENT_NEGOTIATION_CLASS':
'myapp.negotiation.IgnoreClientContentNegotiation',
}
```

默认值: `'rest_framework.negotiation.DefaultContentNegotiation'`

也可以在视图或视图集中使用 `content_negotiation_class` 类属性，指定视图使用的内容协商方法：

```python
from myapp.negotiation import IgnoreClientContentNegotiation
from rest_framework.response import Response
from rest framework.views import APIView

class NoNegotiationView(APIView):
    """
    An example view that does not perform content negotiation.
    """
    content_negotiation_class = IgnoreClientContentNegotiation

    def get(self, request, format=None):
        return Response({
            'accepted media type': request.accepted_renderer.media_type
        })
```