

DRF框架

一、开发模式和RESTful

1、web应用开发模式

1.1、前后端不分离

1.2、前后端分离

2、RESTful介绍

RESTful是目前最流行的API设计风格，REST指的是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是RESTful。

2.1、RESTFUL特点：

- 1、每一个URI代表1种资源；
- 2、客户端使用GET、POST、PUT、DELETE4个表示操作方式的动词对服务端资源进行操作：
 - GET 用来获取资源
 - POST 用来新建资源
 - PUT 用来更新资源
 - DELETE 用来删除资源
- 3、通过操作资源的表现形式来操作资源；
- 4、资源的形式是XML或者json；
- 5、客户端与服务端之间的交互在请求之间是无状态的，从客户端到服务端的每个请求都必须包含理解请求所必需的信息。

3、RESTful API设计风格

1、HTTP动词

对于资源的具体操作类型，由HTTP动词表示。常用的HTTP动词有下面四个（括号里是对应的SQL命令）。

- GET (SELECT) : 从服务器取出资源 (一项或多项) 。
- POST (CREATE) : 在服务器新建一个资源。
- PUT (UPDATE) : 在服务器更新资源 (客户端提供改变后的完整资源) 。
- DELETE (DELETE) : 从服务器删除资源。

2、url路径

- url路径尽量用名词表示,而不用动词

- 不推荐:

```
http://api.lemonban.com/getProjects
http://api.lemonban.com/deleteEnvs
```

- 推荐

```
http://api.lemonban.com/Projects
http://api.lemonban.com/Envs
```

路径中的project表示操作的资源

- 不管是单一资源，还是所有资源，路径中名词尽量用复数

- 推荐

```
# 获取单个数据:
GET方法: http://api.lemonban.com/api.lemonban.com/Env/1
# 获取所有数据:
GET方法: http://api.lemonban.com/rest/allEnvs
```

- 不推荐

```
# 获取单个数据:
GET方法: http://api.lemonban.com/api.lemonban.com/Envs/1
# 获取所有数据:
GET方法: http://api.lemonban.com/rest/Envs
```

3、过滤参数

如果接口需要通过条件过滤返回结果，那么过滤的条件参数，应作为查询字符串参数传递。

- 例如:

```
?limit=10: 指定返回记录的数量
?offset=10: 指定返回记录的开始位置。
?page=2&per_page=100: 指定第几页，以及每页的记录数。
?sortby=name&order=asc: 指定返回结果按照哪个属性排序，以及排序顺序。
```

4、返回状态码

客户端的每一次请求，服务器都必须给出回应。回应包括 HTTP 状态码和数据两部分。

HTTP 状态码就是一个三位数，分成五个类别。

- 1xx：相关信息
- 2xx：操作成功
- 3xx：重定向
- 4xx：客户端错误
- 5xx：服务器错误

这五大类总共包含100多种状态码，覆盖了绝大部分可能遇到的情况。每一种状态码都有标准的（或者约定的）解释，客户端只需查看状态码，就可以判断出发生了什么情况，所以服务器应该返回尽可能精确的状态码。

API 不需要 1xx 状态码，下面介绍其他四类状态码的精确含义。

1)、2XX状态码

200 状态码表示操作成功，restful Api设计风格中，不同的方法可以返回更精确的状态码。

- GET: 200 OK
- POST: 201 Created
- PUT: 200 OK
- PATCH: 200 OK
- DELETE: 204 No Content

上面代码中，POST 返回 201 状态码，表示生成了新的资源；DELETE 返回 204 状态码，表示资源已经不存在

• 注意点：

- 202 Accepted 状态码表示服务器已经收到请求，但还未进行处理，会在未来再处理，通常用于异步操作。
- 发生错误时，不应该使用200状态码，

2)、4XX状态码

4xx 状态码表示客户端错误，主要有下面几种。

400 Bad Request：服务器不理解客户端的请求，未做任何处理。

401 Unauthorized：用户未提供身份验证凭据，或者没有通过身份验证。

403 Forbidden：用户通过了身份验证，但是不具有访问资源所需的权限。

404 Not Found：所请求的资源不存在，或不可用。

405 Method Not Allowed：用户已经通过身份验证，但是所用的 HTTP 方法不在他的权限之内。

410 Gone：所请求的资源已从这个地址转移，不再可用。

415 Unsupported Media Type：客户端要求的返回格式不支持。比如，API 只能返回 JSON 格式，但是客户端要求返回 XML 格式。

422 Unprocessable Entity：客户端上传的附件无法处理，导致请求失败。

`429 Too Many Requests`: 客户端的请求次数超过限额。

3)、5XX 状态码

`5xx` 状态码表示服务端错误。一般来说, API 不会向用户透露服务器的详细信息, 所以只要两个状态码就够了

`500 Internal Server Error`: 客户端请求有效, 服务器处理时发生了意外。

`503 Service Unavailable`: 服务器无法处理请求, 一般用于网站维护状态

5、返回内容

服务器返回的数据格式, 应该尽量使用JSON

针对不同操作, 服务器向用户返回的结果应该符合以下规范。

- GET /collection: 返回资源对象的列表 (数组)
- GET /collection/1: 返回单个资源对象
- POST /collection: 返回新生成的资源对象
- PUT /collection/resource: 返回完整的资源对象
- DELETE /collection/resource: 返回一个空文档

二、DRF框架介绍

Django REST framework 框架是一个用于构建Web API 的强大而又灵活的工具。

通常简称为DRF框架 或 REST framework。

DRF框架是建立在Django框架基础之上, 由Tom Christie大牛二次开发的开源项目。

• 特点

- 提供了定义序列化器Serializer的方法, 可以快速根据 Django ORM 或者其它库自动序列化/反序列化;
- 提供了丰富的类视图、Mixin扩展类, 简化视图的编写;
- 丰富的定制层级: 函数视图、类视图、视图集合到自动生成 API, 满足各种需要;
- 多种身份认证和权限认证方式的支持;
- 内置了限流系统;
- 直观的 API web 界面;
- 可扩展性, 插件丰富

DRF (Django REST framework) 框架是建立在Django框架基础之上, 是一个用于构建Web API 的强大而又灵活的工具, 通常简称为DRF框架 或 REST framework。

1、安装和使用

• 安装DRF

```
pip install djangorestframework
```

- **注册rest_framework应用**

我们利用在Django框架学习中创建的demo工程，在settings.py的INSTALLED_APPS中添加'rest_framework'。

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

2、Django开发RESTful接口

- 1、定义模型类
- 2、定义路由
- 3、定义视图

3、DRF开发RESTful接口

- 1、定义模型类
- 2、定义序列化器
- 3、定义路由
- 4、定义视图

三、序列化器

- 序列化器的作用：
 - 进行数据的校验
 - 对数据对象进行转换

1、序列化器的定义

Django REST framework中的序列化器通过类来定义，必须继承自rest_framework.serializers.Serializer，序列化器中的字段和模型类中的字段类型保持一致

- 模型类如下：

```
class VipCourseData(models.Model):
    stu_number = models.CharField(max_length=20, default=None,
verbose_name='学习人数')
    sum_number = models.CharField(max_length=20, default=None,
verbose_name='累计人数')
    course_score = models.CharField(max_length=20, default=None,
verbose_name='课程评分')
    price = models.CharField(max_length=40, default=0, verbose_name='课程价
格')
```

- 序列化器的定义

```
class VipCourseDataSerializer(serializers.Serializer):
    stu_number = serializers.CharField(max_length=20)
    sum_number = serializers.CharField(max_length=20)
    course_score = serializers.CharField(max_length=20)
    price = serializers.CharField(max_length=40)
```

2、字段类型与选项

常用字段类型：

字段	字段构造方式
BooleanField	BooleanField()
NullBooleanField	NullBooleanField()
CharField	CharField(max_length=None, min_length=None, allow_blank=False, trim_whitespace=True)
EmailField	EmailField(max_length=None, min_length=None, allow_blank=False)
RegexField	RegexField(regex, max_length=None, min_length=None, allow_blank=False)
SlugField	SlugField(max_length=50, min_length=None, allow_blank=False) 正则字段, 验证正则模式[a-zA-Z0-9-]+
URLField	URLField(max_length=200, min_length=None, allow_blank=False)
IPAddressField	IPAddressField(protocol='both', unpack_ipv4=False, **options)
IntegerField	IntegerField(max_value=None, min_value=None)
FloatField	FloatField(max_value=None, min_value=None)
DecimalField	DecimalField(max_digits, decimal_places, coerce_to_string=None, max_value=None, min_value=None) max_digits: 最多位数 decimal_places: 小数点位置

字段	字段构造方式
DateTimeField	DateTimeField(format=api_settings.DATETIME_FORMAT, input_formats=None)
DateField	DateField(format=api_settings.DATE_FORMAT, input_formats=None)
TimeField	TimeField(format=api_settings.TIME_FORMAT, input_formats=None)
DurationField	DurationField()
ChoiceField	ChoiceField(choices) choices与Django的用法相同
MultipleChoiceField	MultipleChoiceField(choices)
FileField	FileField(max_length=None, allow_empty_file=False, use_url=UPLOADED_FILES_USE_URL)
ImageField	ImageField(max_length=None, allow_empty_file=False, use_url=UPLOADED_FILES_USE_URL)
ListField	ListField(child=, min_length=None, max_length=None)
DictField	DictField(child=)

选项参数：

参数名称	作用
max_length	最大长度
min_lenght	最小长度
allow_blank	是否允许为空
trim_whitespace	是否截断空白字符
max_value	最小值
min_value	最大值

通用参数：

参数名称	说明
read_only	表明该字段仅用于序列化输出，默认False
write_only	表明该字段仅用于反序列化输入，默认False
required	表明该字段在反序列化时必须输入，默认True
default	反序列化时使用的默认值
allow_null	表明该字段是否允许传入None，默认False

参数名称	说明
validators	该字段使用的验证器
error_messages	包含错误编号与错误信息的字典
label	用于HTML展示API页面时，显示的字段名称
help_text	用于HTML展示API页面时，显示的字段帮助提示信息

3、序列化操作

- **序列化 ---> 将python对象 转换为json格式数据**

定义好Serializer类后，如果要通过序列化器类进行序列化，需要先创建Serializer对象了。

Serializer的参数为：

```
Serializer(instance=None, data=empty, **kwargs)
```

说明：

- 1) 用于序列化时，将模型类对象传入**instance**参数
- 2) 用于反序列化时，将要被反序列化的数据传入**data**参数
- 3) 除了instance和data参数外，在构造Serializer对象时，还可通过**context**参数额外添加数据，如

```
serializer = AccountSerializer(account, context={'request': request})
```

通过context参数附加的数据，可以通过Serializer对象的context属性获取。

1、视图中的序列化操作

```
def get(self, request):
    """查询所有数据"""
    cs = VipCourseData.objects.all()
    ser = VipCourseDataSerializer(instance=cs, many=True)
    return JsonResponse(ser.data, safe=False)
```

- 对多个数据进行序列化加参数：many=True

2、关联对象嵌套序列化

对多个数据进行序列化加参数：many=True

- **1、PrimaryKeyRelatedField**

- 返回关联字段的id

```
course = serializers.PrimaryKeyRelatedField()
```


- 2、StringRelatedField

- 返回关联字段模型类 `__str__` 方法返回的内容

```
course = serializers.StringRelatedField()
```

- 3、使用关联对象的序列化器

```
# 关联模型类的序列化器
course = CourseSerializer()
```

4、反序列化操作

- 反序列化 ---> 将json格式数据 转换为python对象

在进行反序列化操作是，会先对象数据进行验证，验证通过的情况下再进行保存

反序列化时，初始化序列化器对象，将要被反序列化的数据传入**data**参数

1、数据验证

- 1、校验数据

- 调用`is_valid()`方法进行验证，验证成功返回True，否则返回False
- 验证成功，可以通过序列化器对象的`validated_data`属性获取数据

```
ser = BookSerializer(data=data)
ser.is_valid()
```

- 注意：`is_valid()`会根据序列化器中字段对应的约束来进行校验

2、常用字段说明

- 1、字段长度和是否校验的参数

字段选项参数前面都列出来了，常用的几个字段

- `max_length`:字段的长度校验
- `min_length`:字段的长度校验
- `required=False`: 不需要校验字段是否为空

- 2、控制序列化和反序列化的字段

- `read_only = True`: 只参与序列号返回，不参与反序列化校验

- `write_only = True`: 只参与反序列化校验，不参与序列号返回

3、自定义验证方法

- 对单一字段进行验证

- 验证方法名的规范: validate_字段名

```
def validate_title(self,value):  
    :params value : 字段值  
    验证处理逻辑
```

- 对多个字段进行验证

- 验证方法名:validate

```
def validate(self,attrs):  
    :params attrs : 包含所有字段的值的（类似字典）  
    验证处理逻辑
```

- 通过validators字段指定验证方法

```
def length_validate(value):  
    if not(10<len(value)<20):  
        raise serializers.ValidationError("字段的长度不在10-20之间")  
  
class VipDataSerializer(serializers.Serializer):  
    stu_number = serializers.CharField(max_length=20,validators=[])  
    sum_number = serializers.CharField(max_length=20)  
    course_score = serializers.CharField(max_length=20)  
    price = serializers.CharField(max_length=40)
```

4、保存数据: save方法

验证通过后，如需保存数据，直接调用序列化器对象的save方法即可，save方法会自动触发序列化器中对应的方法来保存数据

```
# 反序列化  
ser = BookSerializer(data=data)  
# 校验数据  
ser.is_valid()  
# 保存数据  
ser.save()
```

- 注意点:

- 保存: save会调用序列化器中定义create方法

- 更新: save会调用序列化器中定义update方

5、模型序列化器

为了方便我们定义序列化器, DRF为我们提供了ModelSerializer模型类序列化器来帮助我们快速创建一个Serializer类。

ModelSerializer与常规的Serializer相同, 但提供了:

- 基于模型类自动生成一系列字段
- 基于模型类自动为Serializer生成validators, 比如unique_together
- 包含默认的create()和update()的实现

1、模型序列化器的使用

- 定义模型序列化器类, 直接继承于serializers.ModelSerializer即可

```
class VipDataSerializer(serializers.ModelSerializer):
    class Meta:
        model = BookInfo
        fields = '__all__'
```

- 指定模型类和需要序列化得字段
 - model 指明参照哪个模型类
 - fields 指明为模型类的哪些字段生成

2、指定字段

1、fields

- fields = `__all__` 代表模型类中得所有字段都进行序列化
- 也可以通过fields指定具体字段

```
class VipDataSerializer(serializers.ModelSerializer):
    class Meta:
        model = BookInfo
        # fields = '__all__'    # 所有字段
        fields = ('id', 'tu_number', 'sum_number') # 指定序列化得字段
```

2、exclude

- 使用exclude可以明确排除掉哪些字段

```
class VipDataSerializer(serializers.ModelSerializer):
    class Meta:
        model = BookInfo
        exclude = ('id',) # id字段不参与序列化
```

3、read_only_fields

- 通过read_only_fields可以指明只读的字段

```
class VipDataSerializer(serializers.ModelSerializer):
    class Meta:
        model = BookInfo
        fields = '__all__'
        read_only_fields = ('id',)
```

3、修改字段的参数选项

- 使用extra_kwargs参数为ModelSerializer添加或修改原有的选项参数
 - 通过字段名指定字段对应的参数和值

```
class VipDataSerializer(serializers.ModelSerializer):
    class Meta:
        model = BookInfo
        # fields = '__all__'
        fields = ('id', 'tu_number', 'sum_number')
        extra_kwargs = {
            'tu_numbe': {'min_value': 0, 'required': True},
            'sum_number': {'min_value': 0, 'required': True},
        }
```

四、视图

1、Request对象

REST framework 传入视图的request对象不再是Django默认的HttpRequest对象，而是REST framework提供的扩展了HttpRequest类的Request类的对象。无论前端发送的哪种格式的数据，我们都可以以统一的方式读取数据。

1、data属性

request.data 获取请求体数据。

- 不仅支持表单类型数据，也支持JSON数据，还支持文件数据的获取

2、query_params

request.query_params获取查询字符串参数的，Django的 request.GET 的作用一样

2、Response对象

rest_framework.response.Response

REST framework提供了一个响应类 Response，使用该类构造响应对象时，响应的具体数据内容会被转换（render渲染）成符合前端需求的类型。

```
Response(data, status=None, template_name=None, headers=None, content_type=None)
```

`data` 不能是复杂结构的数据，如Django的模型类对象，对于这样的数据我们可以使用 `serializer` 序列化器序列化处理后（转为了Python字典类型）再传递给 `data` 参数。

1、Response参数说明:

- `data`: 为响应准备的序列化处理后的数据;
- `status`: 状态码，默认200;
- `template_name`: 模板名称，如果使用 `HTMLRenderer` 时需指明;
- `headers`: 用于存放响应头信息的字典;
- `content_type`: 响应数据的Content-Type，通常此参数无需传递，REST framework会根据前端所需类型数据来设置该参数

3、APIView

`rest_framework.views.APIView` 继承django的`View`，在`View`的基础上做了更多的功能扩展。

1、APIView与View的区别:

- 传入到视图方法中的是REST framework的 `Request` 对象
- 视图方法可以返回REST framework的 `Response` 对象
- 任何 `APIException` 异常都会被捕获到，并且处理成合适的响应信息;
- 扩展了身份认证、权限检查、流量控制这三个功能

2、扩展的功能（后面讲）：

- `authentication_classes`：身份认证
- `permission_classes`：权限检查
- `throttle_classes`：限流

4、GenericAPIView

`rest_framework.generics.GenericAPIView` 继承自 `APIView`，增加了对于列表视图和详情视图可能用到的通用支持方法。

1、扩展的类属性:

- `queryset`: 指定当前类视图使用的查询集
- `serializer_class` : 类视图使用的序列化器

2、扩展的方法:

- `self.queryset()`:获取查询集
- `self.serializer()`:获取序列化器
- `self.get_object()`:获取指定的单一对象

3、扩展功能（后面讲）：

- `pagination_class` : 数据分页
- `filter_backends`: 数据过滤&排序
- 指定单一数据获取的参数字段:
 - `lookup_field` 查询单一数据库对象时使用的条件字段，默认为'`pk`'
 - `lookup_url_kwarg` 查询单一数据时URL中的参数关键字名称，默认与`look_field`相同

5、视图扩展类

1、基本扩展类

- `ListModelMixin`:
 - 列表视图扩展类，提供`list`方法快速实现列表视图
 - 返回200状态码
- `CreateModelMixin` :
 - 创建视图扩展类，提供`create`方法快速实现创建资源的视图
 - 成功返回201状态码，如果序列化器对前端发送的数据验证失败，返回400错误
- `RetrieveModelMixin`: 获取单一数据
 - 详情视图扩展类，提供`retrieve`方法，可以快速实现返回一个存在的数据对象。
 - 如果成功，返回200， 否则返回404。

- **UpdateModelMixin: 更新数据**
 - 更新视图扩展类，提供update方法和partial_update方法，可以快速实现更新一个存在的数据对象。
 - 成功返回200，序列化器校验数据失败时，返回400错误。
- **DestroyModelMixin:**
 - 删除视图扩展类，提供destroy方法，可以快速实现删除一个存在的数据对象。
 - 成功返回204，不存在返回404。

2、视图扩展类

- **1、CreateAPIView**
 - 继承自: GenericAPIView、CreateModelMixin
 - 提供 post 方法
- **2、ListAPIView**
 - 继承自: GenericAPIView、ListModelMixin
 - 提供 get 方法
- **3、RetrieveAPIView**
 - 继承自: GenericAPIView、RetrieveModelMixin
 - 提供 get 方法
- **4、DestroyAPIView**
 - 继承自: GenericAPIView、DestroyModelMixin
 - 提供 delete 方法
- **5、UpdateAPIView**
 - 继承自: GenericAPIView、UpdateModelMixin
 - 提供 put 和 patch 方法

- 6、RetrieveUpdateAPIView
 - 继承自: GenericAPIView、RetrieveModelMixin、UpdateModelMixin
 - 提供 get、put、patch方法
- 7、RetrieveUpdateDestoryAPIView
 - 继承自: GenericAPIView、RetrieveModelMixin、UpdateModelMixin、DestoryModelMixin
 - 提供 get、put、patch、delete方法

6、视图集

1、视图集的使用

ViewSet视图集类不再实现get()、post()等方法，而是实现动作 **action** 如 list()、create() 等。将一系列逻辑相关的动作放到一个类中：

- list() 提供一组数据
- retrieve() 提供单个数据
- create() 创建数据
- update() 保存数据
- destory() 删除数据

注意点：

在使用视图集的时候，在配置路由的时候，用自行指定请求方法和处理的视图函数

```
urlpatterns = [
    url(r'^books/$', xxViewSet.as_view({'get': 'list'}),
    url(r'^books/(?P<pk>\d+)/$', XXXnfoViewSet.as_view({'get': 'retrieve'}))
]
```

2、action属性

视图集只在使用as_view()方法的时候，才会将**action**动作与具体请求方式对应上。

3、常用视图集类

1) ViewSet

继承自 `APIView`，作用也与APIView基本类似，提供了身份认证、权限校验、流量管理等。

在ViewSet中，没有提供任何动作action方法，需要我们自己实现action方法。

2) GenericViewSet

继承自 `GenericAPIView`，作用也与 `GenericAPIView` 类似，提供了 `get_object`、`get_queryset` 等方法便于列表视图与详情信息视图的开发。

3) ModelViewSet

继承自 `GenericAPIView`，同时包括了 `ListModelMixin`、`RetrieveModelMixin`、`CreateModelMixin`、`UpdateModelMixin`、`DestroyModelMixin`。

4) ReadOnlyModelViewSet

继承自 `GenericAPIView`，同时包括了 `ListModelMixin`、`RetrieveModelMixin`。

7、路由

对于视图集 `ViewSet`，我们除了可以自己手动指明请求方式与动作 `action` 之间的对应关系外，还可以使用 `Routers` 来帮助我们快速实现路由信息。

`REST framework` 提供了两个 `router`

- **SimpleRouter (推荐)**
- **DefaultRouter (不推荐)**
- **DefaultRouter与SimpleRouter的区别是，DefaultRouter会多附带一个默认的API根视图，返回一个包含所有列表视图**

1、创建router对象并注册

```
from rest_framework import routers

router = routers.SimpleRouter()
router.register(r'vips', BookInfoViewSet)
```

- **register(prefix, viewset, base_name)**
 - **prefix** 该视图集的路由前缀
 - **viewset** 视图集
 - **base_name** 路由名称的前缀

如上述代码会形成的路由如下：

```
^vips/$
^vip/{pk}/$
```

2、添加路由数据

```
urlpatterns = [
    ...
]
urlpatterns += router.urls
```

五、其他功能

1、认证&权限

1、认证

1)、全局配置

- 在配置文件中配置全局默认认证方案

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.BasicAuthentication', # Basic认证
        'rest_framework.authentication.SessionAuthentication', # session认证
    )
}
```

2)、单个视图配置

在视图中通过设置authentication_classes属性来设置视图的认证方案

```
from rest_framework.authentication import SessionAuthentication,
BasicAuthentication
from rest_framework.views import APIView

class VIPView(APIView):
    # 指定认证的方式
    authentication_classes = (SessionAuthentication, BasicAuthentication)
```

认证失败会有两种可能的返回值：

- 401 Unauthorized 未认证
- 403 Permission Denied 权限被禁止

2、权限

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

1)、全局权限管理

在配置文件中设置默认的权限管理类，

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

如果未指明，默认采用如下默认配置（所有用户均可访问）

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

2)、单个视图权限

在视图通过permission_classes属性来设置权限，

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

class VIPView(APIView):
    permission_classes = (IsAuthenticated,)
    ...
```

3)、权限选项

- 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

2、限流

对接口访问的频次进行限制，以减轻服务器压力(反爬虫的一种手段)。

1、限流类型

- 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
- 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
- ScopedRateThrottle
 - 限制用户对于具体视图的访问频次，通过ip或user id。
 - 视图中使用throttle_scope 指定频次

2、全局配置

- DEFAULT_THROTTLE_CLASSES:设置限流类型
- DEFAULT_THROTTLE_RATES: 设置限制的频次

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle'
    ],
    'DEFAULT_THROTTLE_RATES': {
        'anon': '100/day', # 未认证的用户，每天100次
        'user': '1000/day' # 认证的用户，每天1000次
    }
}
```

- 频率周期
 - second: 每秒
 - minute: 每分钟
 - hour: 每小时
 - day: 每天

3、局部配置

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_RATES': {
        'user': '10/day'
    }
}
```

也可以在具体视图中通过throttle_classes属性来指定限流的类型

```
from rest_framework.throttling import UserRateThrottle
from rest_framework.views import APIView

class ExampleView(APIView):
    # 类视图中指定限流类型
    throttle_classes = (UserRateThrottle,)
```

3、过滤

对于列表数据可能需要根据字段进行过滤，我们可以通过添加django-filter扩展来增强支持。

```
pip install django-filter
```

在配置文件中增加过滤后端的设置：

```
# 注册应用，
INSTALLED_APPS = [
    ...
    'django_filters',
]
# 指定过滤器
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS':
    ('django_filters.rest_framework.DjangoFilterBackend',)
}
```

在视图中添加filterset_fields属性，指定可以过滤的字段

```
class StudentView(ListAPIView):
    queryset = BookInfo.objects.all()
    serializer_class = BookInfoSerializer
    filterset_fields = ('age',)

# 127.0.0.1:8000/students/?age=18
```

4、排序

对于列表数据，REST framework提供了OrderingFilter过滤器来帮助我们快速指明数据按照指定字段进行排序。

- **filter_backends:**指定排序过滤器
- **ordering_fields:** 指定排序的可选字段

REST framework会在请求的查询字符串参数中检查是否包含了ordering参数，如果包含了ordering参数，则按照ordering参数指明的排序字段对数据集进行排序。

```
from rest_framework.filters import OrderingFilter

class StudentView(ListAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    filter_backends = [OrderingFilter] # 指定排序的过滤器
    ordering_fields = ('age', 'id' ) # 指定排序的字段

# url 指明通过age字段排序
# 127.0.0.1:8000/students/?ordering=age
# url 指明通过id字段排序
# 127.0.0.1:8000/students/?ordering=id
```

5、分页

REST framework提供了分页的支持。

1、全局配置

在配置文件中设置全局的分页方式：

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

2、局部配置

在不同的视图中可以通过pagination_class属性来指定不同的分页器

- 自定义分页器

定义一个继承PageNumberPagination的类型，在子类中通过属性定义分页器的数据：

- page_size 每页默认的数据条数
- page_query_param 前端发送的页数关键字名，默认为"page"
- page_size_query_param 前端发送的每页数关键字名，默认为None
- max_page_size 每页最多的数据条数

```
class StuPagination(PageNumberPagination):  
    # 默认每页数据量  
    page_size = 20  
    page_size_query_param = 'page_size'  
    # 每页的数据量的最大值  
    max_page_size = 10000
```

- 使用分页器

```
class StuView(RetrieveAPIView):  
    queryset = Students.objects.all()  
    serializer_class = StudentsSerializer  
    pagination_class = StuPagination
```

- 关闭分页功能

如果在视图内关闭分页功能，只需在视图内设置**

```
pagination_class = None
```

3、分页器类型

1) PageNumberPagination

- 前端访问网址形式:

```
http://127.0.0.1:8000/students/?page=4
```

- 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

2) LimitOffsetPagination

- 前端访问网址形式:

```
http://127.0.0.1:8000/students/?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

6、异常处理

REST framework提供了异常处理, 如果没有自定义默认会采用默认的处理方法方式

```
REST_FRAMEWORK = {  
    # REST framework中默认异常处理方法  
    'EXCEPTION_HANDLER': 'rest_framework.views.exception_handler'  
}
```

1、自定义异常处理的方法

- 1、定义异常处理的方法

```
from rest_framework.views import exception_handler

def custom_exception_handler(exc, context):
    # Call REST framework's default exception handler first,
    # to get the standard error response.
    response = exception_handler(exc, context)

    # Now add the HTTP status code to the response.
    if response is not None:
        response.data['status_code'] = response.status_code

    return response
```

- 2、在配置文件中指定自定义的异常处理

```
REST_FRAMEWORK = {
    'EXCEPTION_HANDLER': 'project.app.except_handle.exception_handle'
}
```

2、REST framework定义的异常

- APIException 所有异常的父类
- ParseError 解析错误
- AuthenticationFailed 认证失败
- NotAuthenticated 尚未认证
- PermissionDenied 权限决绝
- NotFound 未找到
- MethodNotAllowed 请求方式不支持
- NotAcceptable 要获取的数据格式不支持
- Throttled 超过限流次数
- ValidationError 校验失败

7、文件上传

```
#模型
class UploadFile(models.Model):
    """文件上传"""
    file = models.FileField()
    path = models.CharField(max_length=200, blank=True)

    def __str__(self):
        return self.path

    class Meta:
        db_table = 'upload_file'
        verbose_name_plural = "文件上传"

# 序列化器
class UploadFileSerializer(serializers.ModelSerializer):
    """文件上传"""
```



```

class Meta:
    model = UploadFile
    fields = '__all__'

# 视图
class UpFileAPIView(ModelViewSet):
    """文件上传"""
    serializer_class = UploadFileSerializer
    queryset = UploadFile.objects.all()

    def create(self, request, *args, **kwargs):
        res = super().create(request, *args, **kwargs)
        obj = UploadFile.objects.get(id=res.data['id'])
        obj.path = obj.file.name
        obj.save()
        res.data['path'] = obj.name
        return res

    def destroy(self, request, *args, **kwargs):
        """文件删除"""
        res = super().destroy(request, *args, **kwargs)
        # 删除保存的文件
        os.remove('files/{}'.format(self.get_object().name))
        return res

```

8、接口文档

REST framework通过第三方库可以自动帮助我们生成网页版的接口文档，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

- 在项目路由中添加接口文档的路由，配置如下：

```

from rest_framework.documentation import include_docs_urls

urlpatterns = [
    re_path(r'^docs/', include_docs_urls(title='接口文档'))
]

```

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

- 注册接口文档

- ```
INSTALLED_APPS= [
 ...,
]
```

加上配置

```
REST_FRAMEWORK = {
 # 接口文档配置
 'DEFAULT_SCHEMA_CLASS': 'rest_framework.schemas.coreapi.AutoSchema',
}
```

### 3、文档接口说明

- 1) 单一方法的视图, 可直接使用类视图的文档字符串

```
class StudentsView(generics.ListAPIView):
 """
 返回所有学生信息。
 """
```

- 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

```
class StudentsListCreateView(generics.ListCreateAPIView):
 """
 get:
 返回所有学生信息。

 post:
 添加学生
 """
```

- 3) 对于视图集ViewSet, 仍在类视图的文档字符串中分开定义

注意点: 视图集ViewSet中的retrieve名称, 在接口文档网站中叫做read

```
class BookInfoViewSet(mixins.ListModelMixin, mixins.RetrieveModelMixin,
GenericViewSet):
 """
 list:
 获取所有学生数据

 retrieve:
 获取一个学生
 """
```

## 4、添加参数描述信息

参数的描述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 XXX(models.Model):
 age = models.IntegerField(default=0, verbose_name='年龄', help_text='年龄')
```

## 六、ajax跨域

针对于前后端分离的项目，前端和后台是分开部署的，因此服务端要支持 `CORS` (跨域资源共享) 策略，需要在响应头中加上 `Access-Control-Allow-Origin: *`。

| 位置   | 域名             |
|------|----------------|
| 前端服务 | 127.0.0.1:8080 |
| 后端服务 | 127.0.0.1:8000 |

前端与后端分别是不同的端口，这就涉及到跨域访问数据的问题，因为浏览器的同源策略，默认是不支持两个不同域名间相互访问数据，而我们需要在两个域名间相互传递数据，这时我们就要为后端添加跨域访问的支持。\*\*

### from chat-gpt

要在Django REST framework中配置AJAX跨域请求，你可以按照以下步骤进行操作：

1. 安装额外的包：在终端中运行以下命令安装 `django-cors-headers` 包：

```
pip install django-cors-headers
```

2. 在Django项目的 `settings.py` 文件中，将 `corsheaders` 添加到 `INSTALLED_APPS` 设置中：

```
INSTALLED_APPS = [
 # 其他应用程序
 'corsheaders',
]
```

3. 在 `settings.py` 文件最下方，添加以下配置：

```
MIDDLEWARE = [
 # 其他中间件
 'corsheaders.middleware.CorsMiddleware',
 # 其他中间件
]

CORS_ORIGIN_ALLOW_ALL = True
```

上述配置将在Django中启用CORS中间件，并允许所有来源进行跨域请求。

如果你想只允许特定的来源进行跨域请求，可以将 `CORS_ORIGIN_ALLOW_ALL` 设置为 `False`，然后添加允许的来源列表：

```
CORS_ORIGIN_ALLOW_ALL = False

CORS_ORIGIN_WHITELIST = [
 'http://example.com',
 'https://example.com',
]
```

在上面的例子中，只有 `http://example.com` 和 `https://example.com` 这两个来源可以进行跨域请求。

4. 保存并重新启动Django开发服务器。

现在，你可以在Django REST framework中使用AJAX进行跨域请求。请记住，跨域请求可能会涉及到安全问题，因此你应该仅允许受信任的来源进行跨域访问，并在确保安全的情况下进行配置。

## 1、django-cors-headers

- 安装

```
pip install django-cors-headers
```

- 添加应用

```
INSTALLED_APPS = (
 ...
 'corsheaders',
 ...
)
```

- 中间键设置

```
MIDDLEWARE = [
 'corsheaders.middleware.CorsMiddleware',
 ...
]
```

- 添加白名单



- 加密后的头部

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
```

## 2)、payload

载荷是 JSON Web Token 的主体内容部分，里面存放一些有效信息，JSON Web Token 标准定义了几个标准字段：

- iss: 该JWT的签发者
- sub: 该JWT所面向的用户
- aud: 接收该JWT的一方
- exp: 什么时候过期，这里是一个Unix时间戳
- at: 在什么时候签发的

除了标准定义中的字段外，我们还可以自定义字段，比如在 JWT 中，我们的载荷信息可能如下

```
{
 "sub": "lemonban01",
 "name": "musen",
 "admin": true,
 "exp": 12132323423423
}
```

然后将其进行base64加密，得到JWT的第二部分。

```
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjYWRtaW4iOnRydWV9
```

## 3)、signature

签名是 JSON Web Token 中比较重要的一部分，前面两部分都是使用 Base64 进行编码的，signature 需要使用编码后的 header 和 payload 以及我们提供的一个密钥，然后使用 header 中指定的签名算法（HS256）进行签名，签名的作用是保证 JWT 没有被篡改过。

JWT的第三部分签证信息由三部分组成：

- header :(base64后的)
- payload :(base64后的)
- secret: 私钥

加密后的header +加密后的payload 结合私钥secret，用加密算法加密，得到最后的签名

## 2、RESTframework JWT

关于签发和核验JWT，我们可以使用Django REST framework JWT扩展来完成。

- 环境安装

```
pip install djangorestframework-simplejwt -i
https://pypi.tuna.tsinghua.edu.cn/simple
```

- settings配置

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

JWT_AUTH = {
 # 指明token的有效期
 'JWT_EXPIRATION_DELTA': datetime.timedelta(days=1),
}
```

## 3、账号登录

### 1. 业务说明

验证用户名和密码，验证成功后，为用户签发JWT，前端将签发的JWT保存下来。

### 2. 后端接口设计

**请求方式：** POST /login/

**请求参数：** JSON 或 表单

| 参数名      | 类型  | 是否必须 | 说明  |
|----------|-----|------|-----|
| username | str | 是    | 用户名 |
| password | str | 是    | 密码  |

**返回数据：** JSON

```
{
 "username": "python",
 "user_id": 1,
 "token":
 "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjo5LCJ1c2VybmFtZSI6InB5dGhvbGgiLCJleHAiOiE1Mjg5ODI2MzQsImVtYWlsIjoieIn0.ejjvVExrBvbp18QIjQbL1TFE0c0ejQgizui_AR0IAU"
}
```

| 返回值      | 类型  | 是否必须 | 说明     |
|----------|-----|------|--------|
| username | str | 是    | 用户名    |
| id       | int | 是    | 用户id   |
| token    | str | 是    | 身份认证凭据 |

### 3. 后端实现

Django REST framework JWT提供了登录签发JWT的视图，可以直接使用

```
from rest_framework_jwt.views import obtain_jwt_token

urlpatterns = [
 url(r'^login/$', obtain_jwt_token),
]
```

但是默认返回值仅有token，我们还需在返回值中增加username和user\_id。

通过修改该视图的返回值可以完成我们的需求。

在项目中自定义一个模块，创建

```
def jwt_response_data(token, user=None, request=None):
 """
 自定义jwt认证成功返回数据
 """
 return {
 'token': token,
 'id': user.id,
 'username': user.username
 }
```

修改配置文件

```
JWT配置
JWT_AUTH = {
 'JWT_EXPIRATION_DELTA': datetime.timedelta(days=1),
 'JWT_RESPONSE_PAYLOAD_HANDLER': '自定义认证类模块路径.jwt_response_data',
}
```



## 4. 增加支持管理员用户登录账号

JWT扩展的登录视图，在收到用户名与密码时，也是调用Django的认证系统中提供的`authenticate()`来检查用户名与密码是否正确。

我们可以通过修改Django认证系统的认证后端（主要是`authenticate`方法）来支持登录账号既可以是用户名也可以是手机号。

**修改Django认证系统的认证后端需要继承 `django.contrib.auth.backends.ModelBackend`，并重写`authenticate`方法。**

`authenticate(self, request, username=None, password=None, **kwargs)` 方法的参数说明：

- request 本次认证的请求对象
- username 本次认证提供的用户账号
- password 本次认证提供的密码

**我们想要让管理员用户才能登录我们的admin后台,这时我们就要修改django原有的用户验证方法。**

重写`authenticate`方法的思路：

1. 根据username参数查找用户User对象，在查询条件中在加上`is_staff=True`的条件
2. 若查找到User对象，调用User对象的`check_password`方法检查密码是否正确

在`utils/authenticate.py`中编写：

```
from django.contrib.auth.backends import ModelBackend
import re
from users.models import User

class MusenModelBackend(ModelBackend):
 def authenticate(self, request, username=None, password=None, **kwargs):
 # 判断是否通过vue组件发送请求
 if request is None:
 try:
 user = User.objects.get(username=username, is_staff=True)
 except:
 return None
 # 判断密码
 if user.check_password(password):
 return user

 else:
 try:
 user = User.objects.get(username=username)
 except:
 try:
 user = User.objects.get(mobile=username)
 except:
 return None

 # 判断密码
 if user.check_password(password):
 return user
 else:
 return None
```

---

在settings.py文件中告知Django使用我们自定义的认证后端

```
AUTHENTICATION_BACKENDS = [
 '自定义认证类模块路径.MusenModeBackend',
]
```

## 前端保存token

前端可以将JWT保存在浏览器的本地存储里，浏览器的本地存储提供了sessionStorage 和 localStorage 两种：

- **sessionStorage** 浏览器关闭即失效
- **localStorage** 长期有效