django-rest-framework

一、序列化

序列化可以把查询集和模型对象转换为json、xml或其他类型，也提供反序列化功能。，也就是把转换后的类型转换为对象或查询集。

REST框架中的序列化程序与Django `Form`和`ModelForm`类的工作方式非常相似。我们提供了一个`Serializer`类，它为您提供了一种强大的通用方法来控制响应的输出，以及一个`ModelSerializer`类，它提供了一个有用的快捷方式来创建处理模型实例和查询集的序列化程序。

1.1声明序列化器

```
from rest_framework import serializers

class xxxSerializer(serializers.Serializer):
    email = serializers.EmailField()
    content = serializers.CharField(max_length=200)
    created = serializers.DateTimeField()
```

1.2 常用field类

- 核心参数

参数名	缺省值	说明
read_only	False	只读，创建或更新对象时不能设置
required	True	如果在反序列化期间未提供字段，则会引发错误。如果在反序列化期间不需要此字段，则设置为false。
default		缺省值，部分更新时不支持
allow_null	False	允许为空
validators	[]	验证器，一个函数列表，验证通不过引起serializers.ValidationError
error_messages	{}	一个错误信息字典

- 常用字段

字段名	说明
BooleanField	对应于django.db.models.fields.BooleanField
CharField	CharField(max_length=None, min_length=None, allow_blank=False, trim_whitespace=True) max_length - 验证输入包含的字符数不超过此数量。 min_length - 验证输入包含不少于此数量的字符。 allow_blank- 如果设置为True则则应将空字符串视为有效值。如果设置为False则则空字符串被视为无效并将引发验证错误。默认为False。 trim_whitespace- 如果设置为Truethen则会修剪前导和尾随空格。默认为True
EmailField	EmailField(max_length=None, min_length=None, allow_blank=False)
IntegerField	IntegerField(max_value=None, min_value=None) max_value 验证提供的数字是否不大于此值。 min_value 验证提供的数字是否不低于此值。
FloatField	FloatField(max_value=None, min_value=None)
DateTimeField	DateTimeField(format=api_settings.DATETIME_FORMAT, input_formats=None, default_timezone=None) format格式字符串可以是显式指定格式的Python strftime格式 input_formats - 表示可用于解析日期的输入格式的字符串列表

- 实例

```
#models.py
class User(models.Model):
    username = models.CharField(max_length=30)
    password_hash =
models.CharField(max_length=20,db_column='password')
    age = models.IntegerField(default=0)

    class Meta:
        db_table = 'user'


#serializers.py
def validate_password(password):
    if re.match(r'\d+$',password):
        raise serializers.ValidationError("密码不能是纯数字")

class UserSerializer(serializers.Serializer):
    # id = serializers.IntegerField()
    username = serializers.CharField(max_length=30)
    password_hash = serializers.CharField(min_length=3,validators=
[validate_password])
    age = serializers.IntegerField()

    def create(self, validated_data):
        return User.objects.create(**validated_data)
    def update(self, instance, validated_data):
        instance.username =
validated_data.get('username',instance.username)
        instance.password_hash =
validated_data.get('password_hash',instance.password_hash)
        instance.age = validated_data.get('age',instance.age)
        instance.save()
        return instance
```

```
#view.py
class ExampleView(APIView):
    def get(self,request):
        #序列化
        user = User.objects.get(pk=1)
        serializer = UserSerializer(instance=user)
        print(serializer.data)
        return Response(serializer.data)
```

1.3 ModelSerializer

ModelSerializer类能够让你自动创建一个具有模型中相应字段的Serializer类。这个ModelSerializer类和常规的Serializer类一样，不同的是：

- 它根据模型自动生成一组字段。
- 它自动生成序列化器的验证器，比如unique_together验证器。
- 它默认简单实现了.create()方法和.update()方法。

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        #你还可以将fields属性设置成'__all__'来表明使用模型中的所有字段。
        fields = ('id', 'account_name', 'users', 'created')
```

二、Request和Response

- Request

  REST框架引入了一个扩展了常规HttpRequest的Request对象，并提供了更灵活的请求解析。Request对象的核心功能是request.data属性，它与request.POST类似，但对于使用Web API更为有用。

  ```
  request.POST  # 只处理表单数据  只适用于'POST'方法
  request.data  # 处理任意数据  适用于'POST', 'PUT'和'PATCH'方法
  ```

  - query_params 查询参数

    ```
    request.query_params
    ```

- .method request.method 返回请求的HTTP方法的 大写 字符串表示形式。
- 自定义解析器 REST framework的请求对象提供灵活的请求解析，允许你以与通常处理表单数据相同的方式使用JSON数据或其他媒体类型处理请求。

  可以使用DEFAULT_PARSER_CLASSES设置全局默认的解析器集。例如，以下设置将仅允许具有JSON内容的请求，而不是JSON或表单数据的默认值。

```
REST_FRAMEWORK = {
    'DEFAULT_PARSER_CLASSES': (
        'rest_framework.parsers.JSONParser',
    )
}
```

你还可以设置用于单个视图或视图集的解析器， 使用APIView类视图。

```
    from rest_framework.parsers import JSONParser
    from rest_framework.response import Response
    from rest_framework.views import APIView
    class ExampleView(APIView):
        """
        可以接收JSON内容POST请求的视图。
        """
        parser_classes = (JSONParser,)
        def post(self, request, format=None):
            return Response({'received data': request.data})
```

或者，如果你使用基于方法的视图的@api_view装饰器。

~~~
```
from rest_framework.decorators import api_view
from rest_framework.decorators import parser_classes

@api_view(['POST'])
@parser_classes((JSONParser,))
def example_view(request, format=None):
    """
    可以接收JSON内容POST请求的视图
    """
    return Response({'received data': request.data})
```
~~~

```
~~~
```

- Response

  REST框架还引入了一个Response对象，这是一种获取未渲染（unrendered）内容的TemplateResponse类型，并使用内容协商来确定返回给客户端的正确内容类型。

  ```
  Response(data, status=None, template_name=None, headers=None,
  content_type=None)
  return Response(data)  # 渲染成客户端请求的内容类型。
  ```

  与常规的 HttpResponse 对象不同，你不能使用渲染内容来实例化一个 Response 对象，而是传递未渲染的数据，包含任何Python基本数据类型。

  Response 类使用的渲染器无法自行处理像 Django model 实例这样的复杂数据类型，因此你需要在创建 Response 对象之前将数据序列化为基本数据类型。

  你可以使用 REST framework的 Serializer 类来执行此类数据的序列化，或者使用你自定义的来序列化。

- 状态码（Status codes）

  在你的视图（views）中使用纯数字的HTTP 状态码并不总是那么容易被理解。而且如果错误代码出错，很容易被忽略。REST框架为status模块中的每个状态代码（如HTTP_400_BAD_REQUEST）提供更明确的标识符。使用它们来代替纯数字的HTTP状态码是个很好的主意。

- wrapping

  REST框架提供了两个可用于编写API视图的包装器（wrappers）。

  - 用于基于函数视图的@api_view装饰器。
  - 用于基于类视图的APIView类。

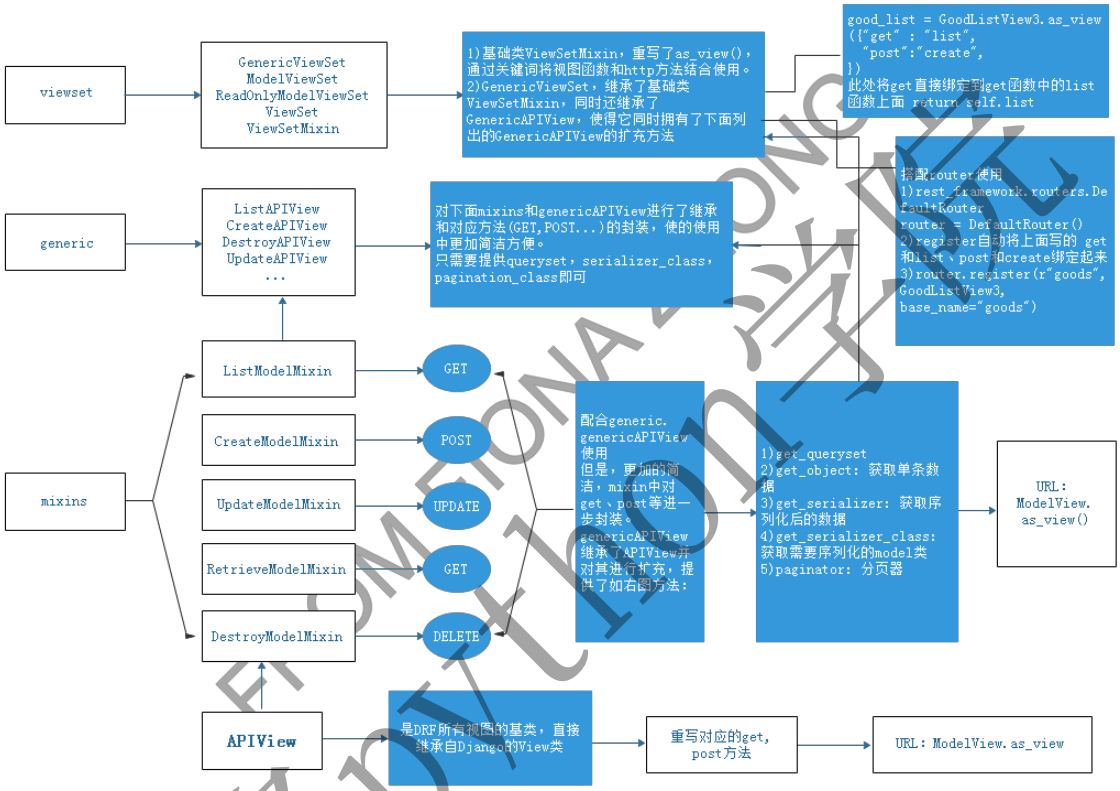
  ```
  from rest_framework.decorators import api_view

  @api_view(['GET', 'POST'])
  def snippet_list(request):
      pass
  ```

三、基于类的视图（CBV）

1.APIView

APIView是DRF的基类，它支持GET POST PUT DELETE等请求类型,且各种类型的请求之间，有了更好的分离在进行dispatch分发之前，会对请求进行身份认证，权限检查，流量控制。



2.mixins

使用基于类的视图的一个最大的好处就是我们可以灵活的选择各种View，使我们的开发更加的简洁。mixins里面对应了ListModelMixin，CreateModelMixin，RetrieveModelMixin，UpdateModelMixin，DestroyModelMixin。

- CreateModelMixin: 定义了创建一个序列对象的方法create(self, request, *args, **kwargs)，保存方法perform_create(self, serializer)，成功获取请求头的方法：get_success_headers(self, data)
- ListModelMixin： 定义了一个获取查询集的方法, many=True：list(self, request, *args, **kwargs)
- RetrieveModelMixin: 定义了一个检索方法，retrieve(self, request, *args, **kwargs)
- UpdateModelMixin： 更新一个模型实例，update(self, request, *args,

**kwargs)

- DestroyModelMixin： 删除一个模型实例，方法destroy(self, request, *args, **kwargs)

3.使用generic

restframework给我们提供了一组混合的generic视图，可以使我们的代码 大大简化。

GenericAPIView：继承自APIView，增加了对列表视图或者详情视图可能用到的通用支持方法。通常使用时，可搭配一个或者多个Mixin扩展类。支持定义的属性：

- 列表视图与详情视图通用：queryset（视图的查询集），serializer_class（视图使用的序列化器）
- 列表视图专用：pagination_classes（分页控制类），filter_backends（过滤控制）
- 详情页视图专用：lookup_field（查询单一数据库对象时使用的条件字段，默认为pk）

提供的方法：

- get_queryset： 获取所有查询集，返回序列器中指定queryset模型的全部对象
- get_serializer_class(self)：返回序列化器，默认返回serializer_class，可以重写
- get_serializer: 获取序列化实例，传入的参数需要通过验证
- get_object：根据传入的查询参数（lookup_url_kwarg or lookup_field)获取查询对象，然后返回， 一般进行联合查询时， 需要重写此方法
- paginate_queryset： 进行分页，返回分页后的单页结果集

提供了5个扩展类：generics.ListAPIView, generics.CreateAPIView, generic.RetrieveAPIView, generic.UpdateAPIView, generic.DestroyAPIView,它们会继承GenericAPIView和mixin对应的视图

四、ViewSet

ViewSet 只是一种基于类的视图，它不提供任何方法处理程序（如 .get() 或.post()）,而是提供诸如 .list() 和 .create() 之类的操作。

一个ViewSet 类当被实例化成一组视图时, 通常会通过使用一个路由类(Router class)来帮你处理复杂的URL定义，最终绑定到一组处理方法。