GeekBrains

Django REST Framework

Serializers. Renderers. Routers

На этом уроке

1. Поймём, для чего в DRF используются serializers, renderers, routers.
2. Узнаем, какие подвиды классов существуют.
3. Поймём, какой вариант выбрать для той или иной задачи.
4. Научимся создавать serializers для разных моделей.
5. Научимся описывать разные типы связей.

Оглавление

Serializers

Введение

Настало время глубже изучить, из каких компонентов состоит DRF. Для чего предназначен тот или иной компонент, и как они взаимодействуют между собой.

На занятии рассмотрим serializers, renderers, parsers и routers. Основное внимание уделим serializers, так как они имеют множество видов и их написание занимает много времени. Renderers и Parsers обычно указываются в настройках всего проекта, а Routes имеют мало подвидов.

Рассмотрим использование serializers сначала на обычных классах python, а затем на классах моделей django.

Для демонстрации разберём следующие классы: авторы, биография авторов, статья, книга. У автора есть имя и год рождения. Биография содержит текст. Статья имеет название, её может написать один автор. У книги есть название, её могут написать несколько авторов.

Этот пример выбран потому, что в нём есть основные виды связей между классами: one-to-one, one-to-many, many-to-many.

Начнём с варианта, который не зависит от django. Код моделей данных выглядит так:

```
class Author:

    def __init__(self, name, birthday_year):
        self.name = name
        self.birthday_year = birthday_year

    def __str__(self):
        return self.name


class Biography:
```

```
    def __init__(self, text, author):
        self.author = author
        self.text = text


class Book:

    def __init__(self, name, authors):
        self.name = name
        self.authors = authors


class Article:

    def __init__(self, name, author):
        self.name = name
        self.author = author
```

Назначение serializers

Ранее мы кратко упоминали о том, что роль serializers — переводить объект django model в json и совершать обратное преобразование из json в объект django model. Это первое приближение, теперь разберём всё детально. Рассмотрим следующий код:

```
from rest_framework import serializers

class AuthorSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=128)
    birthday_year = serializers.IntegerField()

author = Author('Грин', 1880)
serializer = AuthorSerializer(author)
print(serializer.data)  # {'name': 'Грин', 'birthday_year': 1880}
print(type(serializer.data))                                    #                 <class
'rest_framework.utils.serializer_helpers.ReturnDict'>

from rest_framework.renderers import JSONRenderer
renderer = JSONRenderer()
json_bytes = renderer.render(serializer.data)
print(json_bytes)                                                                       #
b'{"name":"\xd0\x93\xd1\x80\xd0\xb8\xd0\xbd","birthday_year":1880}'
print(type(json_bytes))  # <class 'bytes'>

from rest_framework.parsers import JSONParser
stream = io.BytesIO(json_bytes)
data = JSONParser().parse(stream)
print(data)  # {'name': 'Грин', 'birthday_year': 1880}
print(type(data))  # <class 'dict'>
```

```
serializer = AuthorSerializer(data=data)
print(serializer.is_valid())  # True
print(serializer.validated_data)  # OrderedDict([('name', 'Грин'), ('birthday_year',
1880)])
print(type(serializer.validated_data))  # <class 'collections.OrderedDict'>
```

Разберём код по частям:

```
from rest_framework import serializers

class AuthorSerializer(serializers.Serializer):
   name = serializers.CharField(max_length=128)
   birthday_year = serializers.IntegerField()
```

Первым делом мы создали serializer для класса Author. Для этого мы наследуемся от класса serializers.Serializer и определяем нужные типы полей. Таким образом, сериализатор может быть привязан к django-модели или нет. Обратите внимание, что сериализатор напоминает уже знакомые нам Django Forms: Form — не связана с моделью, а ModelForm — связана.

```
author = Author('Грин', 1880)
serializer = AuthorSerializer(author)
print(serializer.data)  # {'name': 'Грин', 'birthday_year': 1880}
print(type(serializer.data))                                    #                <class
'rest_framework.utils.serializer_helpers.ReturnDict'>
```

Далее, если у нас есть объект класса Author, можем передать его в AuthorSerializer и создать объект serializer.

serializer.data — это представление объекта в виде словаря python, который содержит простые типы, доступные в python.

Если проверить type(serializer.data), то увидим специальный тип DRF. Он, в свою очередь, возвращает словарь. Можно опустить этот момент и говорить, что serializer превращает сложный объект в словарь python, содержащий простые типы данных.

Таким образом, ещё раз выделим назначение serializer. ***Serializer преобразует сложный объект в словарь, содержащий простые типы данных, а также выполняет обратное преобразование***.

```
from rest_framework.renderers import JSONRenderer
renderer = JSONRenderer()
json_bytes = renderer.render(serializer.data)
print(json_bytes)                                                                #
```

```
b'{"name":"\xd0\x93\xd1\x80\xd0\xb8\xd0\xbd","birthday_year":1880}'
print(type(json_bytes))  # <class 'bytes'>
```

После того как serializer вернул словарь, нам нужно преобразовать его в независимый от языка python формат. Для этого используется Renderer. Можно применить разные Renderers, например, JSONRenderer. Он преобразует данные в формат json. После создания объекта JSONRenderer() в метод render мы передаём словарь данных и получаем на выходе набор байт, содержащий json. При проверке типа результата type(json_bytes) появятся bytes.

Внимание! *Строго говоря, формат JSON представляет собой текст, а не байты. Приведение в байты сделано для удобства работы самого DRF. Как и в случае с сериализатором, который возвращает rest_framework.utils.serializer_helpers.ReturnDict вместо dict. Концептуально можно говорить, что Render преобразует python-словарь в json.*

Таким образом, ***Renderer преобразует python-словарь, содержащий простые типы данных, в формат данных, не зависимый от python*** (например JSON, HTML, XML…).

```
from rest_framework.parsers import JSONParser
stream = io.BytesIO(json_bytes)
data = JSONParser().parse(stream)
print(data)  # {'name': 'Грин', 'birthday_year': 1880}
print(type(data))  # <class 'dict'>
```

Для обратного преобразования из байт, которые содержат json, используется Parser. Парсер принимает в себя специальный тип <class '_io.BytesIO'>, а на выходе возвращает словарь dict с данными.

```
serializer = AuthorSerializer(data=data)
print(serializer.is_valid())  # True
print(serializer.validated_data)  # OrderedDict([('name', 'Грин'), ('birthday_year', 1880)])
print(type(serializer.validated_data))  # <class 'collections.OrderedDict'>
```

Последний фрагмент кода демонстрирует работу Serializer в обратном направлении. На вход мы подаём данные в виде словаря python и проверяем их на валидность. Если они валидны, на их основании можно восстановить объект Author. Рассмотрим эту возможность далее.

Варианты настройки и использования serializers

Создание и обновление объектов

```
from rest_framework import serializers
```

```
from python_models import Author


class AuthorSerializer(serializers.Serializer):
   name = serializers.CharField(max_length=128)
   birthday_year = serializers.IntegerField()

   def create(self, validated_data):
       return Author(**validated_data)

   def update(self, instance, validated_data):
       instance.name = validated_data.get('name', instance.name)
                       instance.birthday_year   =   validated_data.get('birthday_year',
instance.birthday_year)
       return instance


data = {'name': 'Грин', 'birthday_year': 1880}
serializer = AuthorSerializer(data=data)
serializer.is_valid()
author = serializer.save()

data = {'name': 'Александр', 'birthday_year': 1880}
serializer = AuthorSerializer(author, data=data)
serializer.is_valid()
author = serializer.save()

data = {'birthday_year': 2000}
serializer = AuthorSerializer(author, data=data, partial=True)
serializer.is_valid()
author = serializer.save()
```

Для создания и удаления объектов требуется переопределить методы create и update у serializer.

```
 def create(self, validated_data):
       return Author(**validated_data)
```

В метод create передаётся validated_data — словарь валидных данных, а возвращается объект Author.

```
def update(self, instance, validated_data):
       instance.name = validated_data.get('name', instance.name)
                       instance.birthday_year   =   validated_data.get('birthday_year',
instance.birthday_year)
       return instance
```

В метод update тоже передаётся validated_data, и вместе с ним instance — объект, который мы хотим изменить.

```
data = {'name': 'Грин', 'birthday_year': 1880}
serializer = AuthorSerializer(data=data)
serializer.is_valid()
author = serializer.save()
```

При передаче только словаря данных и вызова метода save будет вызван метод create.

```
data = {'name': 'Александр', 'birthday_year': 1880}
serializer = AuthorSerializer(author, data=data)
```

При передаче объекта author и словаря данных объект изменится и будет вызван метод update.

По умолчанию для обновления объекта нужно передать все поля сериализатора (name и birthday_year). Если же мы хотим передать только несколько полей, а остальные оставить как есть, то нужно использовать параметр partial=True.

```
data = {'birthday_year': 2000}
serializer = AuthorSerializer(author, data=data, partial=True)
```

Важно! *Перед каждым сохранением объекта нужно вызывать метод is_valid serializer-а для проверки данных, иначе возникнет ошибка.*

Ошибки сериализации

```
class AuthorSerializer(serializers.Serializer):
   name = serializers.CharField(max_length=128)
   birthday_year = serializers.IntegerField()

data = {'name': 'Грин', 'birthday_year': 1880}
serializer = AuthorSerializer(data=data)
print(serializer.is_valid())  # True

data = {'name': 'Грин', 'birthday_year': 'abc'}
serializer = AuthorSerializer(data=data)
print(serializer.is_valid())  # False

print(serializer.errors)  # {'birthday_year': [ErrorDetail(string='A valid integer is
required.', code='invalid')]}

serializer.is_valid(raise_exception=True)
```

При передаче в serializer валидных данных метод is_valid вернёт True. Если данные не валидны — False. При невалидных данных возникнет ошибка, если вызвать метод is_valid и передать в него параметр raise_exception=True.

Дополнительная валидация данных

Как и в Django Forms, в Serializers можно добавить дополнительную валидацию по одному или нескольким полям. Рассмотрим следующий пример:

```
class AuthorSerializer(serializers.Serializer):
   name = serializers.CharField(max_length=128)
   birthday_year = serializers.IntegerField()

   def validate_birthday_year(self, value):
       if value < 0:
                   raise  serializers.ValidationError('Год  рождения  не  может  быть
отрицательным')
       return value

   def validate(self, attrs):
       if attrs['name'] == 'Пушкин' and attrs['birthday_year'] != 1799:
           raise serializers.ValidationError('Неверный год рождения Пушкина')
       return attrs
```

Для добавления валидации по некоторому полю в serializer включается метод validate_<имя поля> (validate_birthday_year). Чтобы применить это к нескольким полям — метод validate.

Для генерации ошибки используется класс `serializers.ValidationError`.

Работа с вложенными полями

В предыдущих примерах мы рассматривали serializer для модели данных Author. Её свойствами были поля простых типов: name — строка, birthday_year — целое число. Очень часто на практике встречаются случаи, когда поле одной модели — сложный объект другого класса или список, состоящий из нескольких таких объектов. В нашем случае — это классы Biography, Article, Book. Чаще всего необходимо создать один или несколько serializers для каждого класса и указать нужный Serializer в качестве поля вложенной модели.

Рассмотрим примеры сериализаторов и связей между ними на примере:

```
from rest_framework import serializers
from python_models import Author, Article, Book, Biography


class AuthorSerializer(serializers.Serializer):
   name = serializers.CharField(max_length=128)
   birthday_year = serializers.IntegerField()
```

```
class BiographySerializer(serializers.Serializer):
   text = serializers.CharField(max_length=1024)
   author = AuthorSerializer()


class BookSerializer(serializers.Serializer):
   name = serializers.CharField(max_length=128)
   authors = AuthorSerializer(many=True)


class ArticleSerializer(serializers.Serializer):
   name = serializers.CharField(max_length=128)
   author = AuthorSerializer()


author1 = Author('Грин', 1880)
serializer = AuthorSerializer(author1)
print(serializer.data)  # {'name': 'Грин', 'birthday_year': 1880}

biography = Biography('Текст биографии', author1)
serializer = BiographySerializer(biography)
print(serializer.data)          #     {'text':     'Текст     биографии',     'author':
OrderedDict([('name', 'Грин'), ('birthday_year', 1880)])}

article = Article('Некоторая статья', author1)
serializer = ArticleSerializer(article)
print(serializer.data)          #     {'name':     'Некоторая     статья',     'author':
OrderedDict([('name', 'Грин'), ('birthday_year', 1880)])}

author2 = Author('Пушкин', 1799)
book = Book('Некоторая книга', [author1, author2])  # {'name': 'Некоторая книга',
'authors':     [OrderedDict([('name',     'Грин'),     ('birthday_year',     1880)]),
OrderedDict([('name', 'Пушкин'), ('birthday_year', 1799)])]}

serializer = BookSerializer(book)
print(serializer.data)
```

Рассмотрим код по частям:

```
class AuthorSerializer(serializers.Serializer):
   name = serializers.CharField(max_length=128)
   birthday_year = serializers.IntegerField()
```

AuthorSerializer содержит простые типы полей.

```
class BiographySerializer(serializers.Serializer):
   text = serializers.CharField(max_length=1024)
```

```
    author = AuthorSerializer()

class ArticleSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=128)
    author = AuthorSerializer()
```

В BiographySerializer и ArticleSerializer в поле author мы указали AuthorSerializer. Таким образом, после сериализации получим вложенный словарь с данными автора.

```
class BookSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=128)
    authors = AuthorSerializer(many=True)
```

В BookSerializer мы тоже указали AuthorSerializer, но добавили к нему дополнительный ключ many=True, так как книгу могут написать несколько авторов. После сериализации получим вложенный список, состоящий из словарей с данными автора.

ModelSerializer

Структура моделей данных

Мы рассмотрели основные возможности при использовании Serializer. Теперь разберём дополнительные возможности при работе с django models. Структура моделей для этой же задачи на django имеет следующий вид:

```
from django.db import models


class Author(models.Model):
    name = models.CharField(max_length=32)
    birthday_year = models.PositiveIntegerField()

    def __str__(self):
        return self.name


class Biography(models.Model):
    text = models.TextField()
    author = models.OneToOneField(Author, on_delete=models.CASCADE)


class Book(models.Model):
    name = models.CharField(max_length=32)
    authors = models.ManyToManyField(Author)
```

```
class Article(models.Model):
   name = models.CharField(max_length=32)
   author = models.ForeignKey(Author, models.PROTECT)
```

Настройка ModelSerializer и вложенных полей

ModelSerializer наследуется от Serializer. Это значит, что он обладает всеми возможностями, рассмотренными ранее. Дополнительно он позволяет:

- на основании класса модели создавать нужные типы полей;
- удобно настраивать список полей для отображения;
- самостоятельно сохранять объект модели без переопределения методов create, update;
- выбирать варианты представления вложенных полей.

Рассмотрим все удобные возможности на следующем примере:

```
class AuthorSerializer(serializers.ModelSerializer):
   class Meta:
       model = Author
       fields = '__all__'

class BiographySerializer(serializers.ModelSerializer):
   class Meta:
       model = Biography
       fields = ['text', 'author']

class ArticleSerializer(serializers.ModelSerializer):
   author = AuthorSerializer()

   class Meta:
       model = Article
       exclude = ['name']

class BookSerializer(serializers.ModelSerializer):
   authors = serializers.StringRelatedField(many=True)

   class Meta:
       model = Book
       fields = '__all__'

author1 = Author.objects.create(name='Грин', birthday_year=1880)
serializer = AuthorSerializer(author1)
print(serializer.data)  # {'id': 17, 'name': 'Грин', 'birthday_year': 1880}

biography = Biography.objects.create(text='Некоторая биография', author=author1)
serializer = BiographySerializer(biography)
print(serializer.data)  # {'text': 'Некоторая биография', 'author': 17}

article = Article.objects.create(name='Некоторая статья', author=author1)
```

```
serializer = ArticleSerializer(article)
print(serializer.data)    # {'id': 8, 'author': OrderedDict([('id', 17), ('name',
'Грин'), ('birthday_year', 1880)])}

author2 = Author.objects.create(name='Пушкин', birthday_year=1799)
book = Book.objects.create(name='Некоторая книга')
book.authors.add(author1)
book.authors.add(author2)
book.save()
serializer = BookSerializer(book)
print(serializer.data)    # {'id': 9, 'authors': ['Грин', 'Пушкин'], 'name':
'Некоторая книга'}
```

Рассмотрим код по частям:

```
class AuthorSerializer(serializers.ModelSerializer):
```

Теперь для создания всех serializers используем serializers.ModelSerializer.

```
class Meta:
       model = Author
       fields = '__all__'
```

Минимальные настройки для ModelSerializer:

- модель с которой связан serializer;
- набор полей для отображения.

```
fields = '__all__'
fields = ['text', 'author']
exclude = ['name']
```

Можно указать __all__, и тогда serializer будет отображать все поля модели, включая вложенные. Или указать список полей через запятую. Если вместо fields выбрать exclude, то serializer будет отображать все поля, кроме исключённых.

При настройке сериализации вложенных полей можно использовать следующие возможности:

- указать serializer вложенной модели;
- указать специальный тип сериализатора;
- ничего не указывать.

```
class BiographySerializer(serializers.ModelSerializer):
```

```
    class Meta:
        model = Biography
        fields = ['text', 'author']
```

Модель Biography связана с Author. Но при настройке сериализатора мы ничего не указали. В этом случае при сериализации будет взят id модели Author.

```
biography = Biography.objects.create(text='Некоторая биография', author=author1)
serializer = BiographySerializer(biography)
print(serializer.data)  # {'text': 'Некоторая биография', 'author': 17}
```

```
class ArticleSerializer(serializers.ModelSerializer):
    author = AuthorSerializer()

    class Meta:
        model = Article
        exclude = ['name']
```

Модель Article тоже связана с Author, а в сериализаторе мы указали `author = AuthorSerializer()`. В этом случае за отображение будет отвечать `AuthorSerializer()`, и на выходе появится вложенный словарь.

```
article = Article.objects.create(name='Некоторая статья', author=author1)
serializer = ArticleSerializer(article)
print(serializer.data)    # {'id': 8, 'author': OrderedDict([('id', 17), ('name',
'Грин'), ('birthday_year', 1880)])}
```

```
class BookSerializer(serializers.ModelSerializer):
    authors = serializers.StringRelatedField(many=True)

    class Meta:
        model = Book
        fields = '__all__'
```

Модель Book тоже связана с Author, причём авторов может быть много. В этом случае указан специальный тип поля serializers.StringRelatedField(many=True). Он говорит, что автор будет представлен методом __str__ в модели Author, а ключ many=True позволяет выводить несколько авторов.

```
book = Book.objects.create(name='Некоторая книга')
book.authors.add(author1)
```

```
book.authors.add(author2)
book.save()
serializer = BookSerializer(book)
print(serializer.data)     #  {'id':  9,  'authors':  ['Грин',  'Пушкин'],  'name':
'Некоторая книга'}
```

Метод str-модели Author возвращает имя автора.

Типы связей в ModelSerializer

Все типы связей описаны в разделе Serializer relations, официальной документации DRF. Перечислим основные из них:

1. StringRelatedField — представляет объект методом __str__.
2. PrimaryKeyRelatedField — представляет объект его id (используется по умолчанию).
3. HyperlinkedRelatedField — представляет объект гипперссылкой. Обычно она ведёт на страницу detail этого объекта.
4. SlugRelatedField — позволяет указать несколько полей для отображения объекта.

HyperlinkedModelSerializer

Последний наиболее часто используемый класс для создания serializer — HyperlinkedModelSerializer. Этот класс наследуется от ModelSerializer и позволяет автоматически формировать ссылки для вложенных полей и переходов на detail-страницу объекта. Это удобно и позволяет следовать одному из принципов REST. При обращении к ресурсу в ответе мы имеем ссылки на другие ресурсы.

Для работы с HyperlinkedModelSerializer требуется объект запроса request.

```
from rest_framework import serializers
from .models import Author, Book, Biography, Article


class AuthorSerializer(serializers.HyperlinkedModelSerializer):
   class Meta:
       model = Author
       fields = '__all__'


class BiographySerializer(serializers.HyperlinkedModelSerializer):
   class Meta:
       model = Biography
       fields = '__all__'


class ArticleSerializer(serializers.HyperlinkedModelSerializer):
   class Meta:
       model = Article
```

```
        fields = '__all__'


class BookSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Book
        fields = '__all__'
```

Так выглядит пример работы с типом serializer-a.

Renderers

Настройка и подключение

Выше мы разобрались с назначением Renderers. А теперь рассмотрим варианты их настройки.

Для всего проекта

```
REST_FRAMEWORK = {
    'DEFAULT_RENDERER_CLASSES': [
        'rest_framework.renderers.JSONRenderer',
        'rest_framework.renderers.BrowsableAPIRenderer',
    ]
}
```

В settings.py можно указать Renderers, которые будут использоваться для всего проекта. Если указано несколько renderers, то в зависимости от заголовков (headers) запроса будет выбираться один из них.

Для одной view

Для одной view или viewset можно указать рендер внутри класса.

```
class AuthorViewSet(ModelViewSet):
    renderer_classes = [JSONRenderer, BrowsableAPIRenderer]
    queryset = Author.objects.all()
    serializer_class = AuthorSerializer
```

Или если используется функция, то в качестве декоратора.

```
@api_view(['GET'])
@renderer_classes([JSONRenderer])
def author_view(request, format=None):
...
```

Renderers внутри класса приоритетнее, чем Renderers в settings.py

Виды Renderers

Полный список видов Renderers можно найти в разделе Renderers, официальной документации. Рассмотрим основные из них:

1. JSONRenderer — преобразует данные в формат JSON.
2. TemplateHTMLRenderer — преобразует данные в формат html. Используются html-шаблоны.
3. StaticHTMLRenderer — преобразует данные в html без использования шаблонов, возвращает статическую html-разметку.
4. BrowsableAPIRenderer — преобразует данные для удобной работы с API в брауезере.
5. AdminRenderer — преобразует данные для удобного администрирования.

Можно написать свой вариант Renderer или использовать сторонние библиотеки.

Parsers

Parser похожи на Renderers, но совершают преобразование в обратную сторону, из стороннего формата в формат python.

Их настройка и разновидности аналогичны Renderers и описаны в соответствующем разделе официальной документации.

Routers

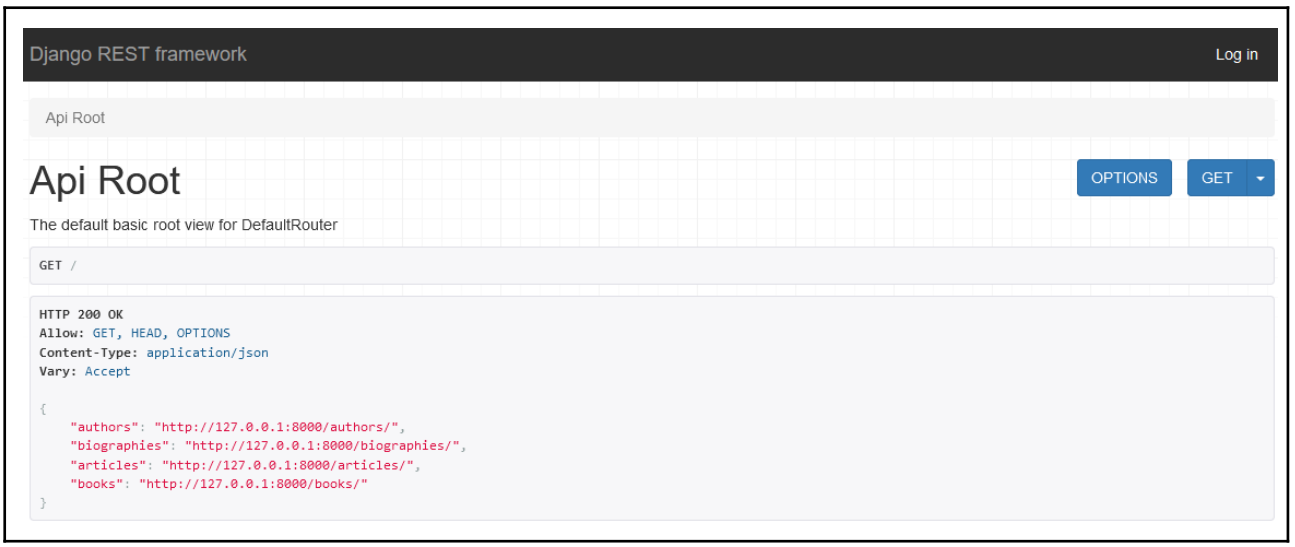
Рассмотрим ещё одну часть DRF — маршрутизаторы (Routers).

Routers позволяют быстро и удобно генерировать адреса нашего REST API.

Существует 2 основные разновидности routers:

- SimpleRouter;
- DefaultRouter.

DefaultRouter наследуется от SimpleRouter. Разница в том, что SimpleRouter генерирует адреса, а DefaultRouter дополнительно создаёт точку входа в API.



Страницу Api Root создал DefaultRouter, при использовании SimpleRouter её не будет.

```
router = DefaultRouter()
router.register('authors', views.AuthorViewSet)
router.register('biographies', views.BiographyViewSet)
router.register('articles', views.ArticleViewSet)
router.register('books', views.BookViewSet)

urlpatterns = [
   path('admin/', admin.site.urls),
   path('api-auth/', include('rest_framework.urls', namespace='rest_framework')),
   path('', include(router.urls)),
]
```

Для использования Router достаточно создать экземпляр нужного класса, и затем зарегистрировать в нём viewset, указав нужный адрес.

И после этого подключить router.urls — адреса, которые создаст Router — в urlpatterns проекта.

```
path('', include(router.urls)),
```

Итоги

В этом занятии мы большую часть времени уделили Serializers, т. к. чаще мы будем работать именно с ними. Познакомились с принципами работы Renderers, Parsers и Routers, для возможности их настройки и изменения.

Глоссарий

1. **Renderer** — часть архитектуры DRF, предназначенная для преобразования словаря python в формат, независящий от python. Обратное преобразование выполняет Parser.
2. **Router** — часть архитектуры DRF, предназначенная для быстрой генерации url-адресов для REST API на основе ViewSet.
3. **Serializer** — часть архитектуры DRF, предназначенная для преобразования сложных объектов в словарь. Он, в свою очередь, содержит простые типы данных. Возможно и обратное преобразование.

Используемые источники и дополнительные материалы

1. Serializers DRF.
2. Serializer fields DRF.
3. Serializer relations DRF.
4. Renderers DRF.
5. Parsers DRF.
6. Routers DRF.

Практическое задание

Создание моделей Project и ToDo. Формирование и настройка API для этих моделей.

В самостоятельной работе тренируем умения

- проектировать модели данных и их связи;
- выбирать и настраивать serializers;
- выбирать и настраивать serializers для связанных полей модели;
- подключить renderers.

Зачем?

Чтобы применять умения при разработке REST API.

Последовательность действий

1. В проекте создать новое приложение для работы с TODO.

2. Добавить модель Project. Это проект, для которого записаны TODO. У него есть название, может быть ссылка на репозиторий и набор пользователей, которые работают с этим проектом. Создать модель, выбрать подходящие типы полей и связей с другими моделями.

3. Добавить модель TODO. Это заметка. У ToDo есть проект, в котором сделана заметка, текст заметки, дата создания и обновления, пользователь, создавший заметку. Содержится и признак — активно TODO или закрыто. Выбрать подходящие типы полей и связей с другими моделями.

4. Создать API для моделей Projects и ToDo. Пока можно использовать ViewSets по аналогии с моделью User.

5. При сериализации моделей выбрать нужный вид для связанных моделей.

6. (Задание со *) На стороне клиента используется camelCase в отличие от snake_case, который мы используем в python. Реализовать представление данных в виде camelCase (https://www.django-rest-framework.org/api-guide/parsers/#camelcase-json).