GeekBrains

Django REST Framework

Views. Filtering. Pagination

На этом уроке

1. Поймём назначение Views.
2. Узнаем, какие варианты создания Views есть в DRF.
3. Научимся добавлять в свои API фильтрацию.
4. Научимся добавлять в свои API постраничный вывод.

Оглавление

Введение

На первом занятии мы уже использовали Viewsets для быстрого создания API для простой модели данных. ModelViewset при использовании по умолчанию создаёт сразу все возможные виды запросов для одной модели. На практике не всегда подходит этот вариант. Часто требуется:

1. Предоставить пользователю только некоторые виды запросов. Например, только get-запрос на просмотр данных.
2. Иметь возможность писать дополнительные действия для конкретной модели данных. Например, менять пароль для пользователя.
3. Настраивать viewset под себя. Например, менять выборку данных, динамически заменять сериализатор и прочее.

DRF предоставляет пользователю все эти возможности. Сам Viewset — это набор Views, который состоит из одной или нескольких View. Они, в свою очередь, могут настраиваться или использоваться по умолчанию.

Рассмотрим варианты View, от самых простых до наиболее комплексных.

Настройка демонстрационного проекта

Все примеры будут приведены для django-проекта с подключённым DRF. В нём имеется приложение mainapp с моделью Article, а сам проект называется blog.

```
from django.db import models
from django.contrib.auth.models import User


class Article(models.Model):
   name = models.CharField(max_length=64)
   text = models.TextField()
   user = models.ForeignKey(User, on_delete=models.CASCADE)
   create = models.DateTimeField()

   def __str__(self):
       return self.name
```

mainapp/models.py

Модель описывает статью, у которой есть название, написавший её пользователь, текст и дата создания.

Serializer для этой модели имеет следующий вид:

```
from rest_framework.serializers import ModelSerializer
from .models import Article
```

```
class ArticleSerializer(ModelSerializer):
   class Meta:
       model = Article
       fields = '__all__'
```

mainapp/serializers.py

В файле mainapp/views.py мы рассмотрим различные виды Views.

Views

APIView

Рассмотрим для начала класс APIView. Это базовый класс для Views в DRF. Он может быть связан с другими частями DRF (например, Renderers) и позволяет полностью самостоятельно написать код обработки того или иного запроса. Например:

```
from rest_framework.views import APIView
from rest_framework.renderers import JSONRenderer
from .models import Article
from rest_framework.response import Response


class ArticleAPIVIew(APIView):
   renderer_classes = [JSONRenderer]

   def get(self, request, format=None):
       articles = Article.objects.all()
       serializer = ArticleSerializer(articles, many=True)
       return Response(serializer.data)
```

mainapp/views.py

Рассмотрим пример по частям.

```
class ArticleAPIVIew(APIView):
   renderer_classes = [JSONRenderer]
```

Создаём свой View, наследуясь от APIView. Свойство renderer_classes позволяет указать список Renderers.

```
def get(self, request, format=None):
       articles = Article.objects.all()
       serializer = ArticleSerializer(articles, many=True)
```

```
        return Response(serializer.data)
```

Метод get отвечает за get-запрос. Мы получаем все статьи, с помощью ArticleSerializer преобразуем выборку в простые типы данных и возвращаем объект Response.

Важно! *Для ответа используется объект класса Respose из DRF, а не из django.*

После того как создали View, нужно указать для него url-адрес. Это можно сделать, добавив объект в набор Views (VIewset). Этот вариант рассмотрим ниже. Или стандартным для django способом, связав View и url в файле url.py.

```
from django.contrib import admin
from django.urls import path
from mainapp import views


urlpatterns = [
   path('views/api-view/', views.ArticleAPIVIew.as_view()),
...
```

blog/urls.py

После перехода по адресу увидим все объекты Article в формате json.

@api_view

DRF позволяет создавать Views не только на классах, но и на функциях.

```
from rest_framework.decorators import api_view, renderer_classes


@api_view(['GET'])
@renderer_classes([JSONRenderer])
def article_view(request):
   articles = Article.objects.all()
   serializer = ArticleSerializer(articles, many=True)
   return Response(serializer.data)
```

mainapp/views.py

В примере мы создали View с такими же опциями, как и в предыдущем, но использовали функцию. Для указания доступных запросов используется декоратор @api_view, а для Renderers — декоратор renderer_classes.

Применение

Базовые View удобно использовать для решения нестандартных задач, которые требуют написания множества уникального кода.

Generic views

GenericAPIView

Следующим уровнем можно считать GenericAPIVIew. Этот класс наследуется от APIView и содержит в себе наиболее общие свойства и методы, такие как queryset, get_queryset и serializer_class. Таким образом, подразумевается, что мы уже работаем с некоторым набором данных и классом для их сериализации.

При добавлении некоторых классов примесей (mixins) и использования GenericAPIView можно получить конкретные классы для той или иной задачи (REST-запроса). Полный список классов можно найти в официальной документации. Рассмотрим основные из них.

Concrete Views

CreateAPIVIew

```
from rest_framework.generics import CreateAPIView


class ArticleCreateAPIView(CreateAPIView):
   renderer_classes = [JSONRenderer]
   queryset = Article.objects.all()
   serializer_class = ArticleSerializer
```

mainapp/views.py

Предоставляет метод post. Для создания модели достаточно указать queryset и serializer_class

ListAPIView

```
from rest_framework.generics import ListAPIView


class ArticleListAPIView(ListAPIView):
   renderer_classes = [JSONRenderer]
   queryset = Article.objects.all()
   serializer_class = ArticleSerializer
```

mainapp/views.py

Предоставляет метод get и выводит список данных из выборки queryset.

RetrieveAPIView

```
from rest_framework.generics import RetrieveAPIView


class ArticleRetrieveAPIView(RetrieveAPIView):
   renderer_classes = [JSONRenderer]
   queryset = Article.objects.all()
   serializer_class = ArticleSerializer
```

mainapp/views.py

Выдаёт метод get и выводит данные об одном объекте из выборки queryset. Для указания адреса требуется параметр pk, чтобы определить id элемента. Например:

```
path('generic/retrieve/<int:pk>/', views.ArticleRetrieveAPIView.as_view())
```

blog/urls.py

DestroyAPIView

```
from rest_framework.generics import DestroyAPIView


class ArticleDestroyAPIView(DestroyAPIView):
   renderer_classes = [JSONRenderer]
   queryset = Article.objects.all()
   serializer_class = ArticleSerializer
```

blog/urls.py

Предоставляет метод delete и удаляет один объект из выборки. В адресе также требуется указать pk объекта.

UpdateAPIView

```
from rest_framework.generics import UpdateAPIView


class ArticleUpdateAPIView(UpdateAPIView):
   renderer_classes = [JSONRenderer]
   queryset = Article.objects.all()
   serializer_class = ArticleSerializer
```

blog/urls.py

Выдаёт методы put и patch для изменения объекта из выборки queryset. Требует pk в url-адресе.

Others

Другие классы — комбинация рассмотренных, например, ListCreateAPIView для списка и создания.

Применение

Эта группа классов применяется практически всегда как по отдельности, так и в наборах views (Viewsets). Она позволяет решать большинство практических задач. Для этого нужно:

1. Выбрать нужный класс для конкретной задачи и типа REST-запроса.
2. Создать свой view путём наследования от этого класса.
3. Связи с url-адресом.
4. При необходимости переопределить доступные методы.

ViewSets

Введение

ViewSets (наборы представлений) позволяют объединять несколько представлений в один набор. Причём можно или описать нужные методы для обработки запросов самостоятельно, или использовать ModelViewSet для создания набора на основе конкретной модели. Можете также собрать ViewSet из нескольких Views. Рассмотрим каждую возможность.

ViewSet

Класс ViewSet в DRF позволяет на его основе создавать набор данных и прописывать важные методы для обработки разных типов запросов. Рассмотрим следующий пример:

```
from rest_framework import viewsets


class ArticleViewSet(viewsets.ViewSet):
   renderer_classes = [JSONRenderer]

   def list(self, request):
       articles = Article.objects.all()
       serializer = ArticleSerializer(articles, many=True)
       return Response(serializer.data)

   def retrieve(self, request, pk=None):
       article = get_object_or_404(Article, pk=pk)
       serializer = ArticleSerializer(article)
```

```
        return Response(serializer.data)
```

mainapp/views.py

В этом примере определены методы list и retrieve. Они соответствуют get-запросам для получения набора данных и информации об одном объекте. Сами методы реализуем сами с помощью модели Article, ArticleSerializer и Response. Это похоже на работу с APIView, но в нашем случае в одном ViewSet можно описать обработку сразу нескольких REST-запросов.

После того как создали Viewset, удобно использовать Router, который сам позаботится о создании адресов для нашего REST API.

```
from django.urls import path, include
from mainapp import views
from rest_framework.routers import DefaultRouter

router = DefaultRouter()
router.register('base', views.ArticleViewSet, basename='article')

urlpatterns = [
   path('viewsets/', include(router.urls)),
...
```

blog/urls.py

В этом примере используется класс DefaultRouter. При регистрации ViewSet — router.register('base', views.ArticleViewSet, basename='article') — указываем точку входа, сам ViewSet и его имя. Затем подключаем urls роутера в urlpatterns.

Важно! *При использовании ViewSet укажите basename при регистрации в роутере. Потому что наш ViewSet не связан с моделью данных и какой-либо выборкой. Нет признака, по которому DRF сам сможет создать название url-адреса.*

Дополнительные действия

Часто требуется добавить некоторые дополнительные действия в наше API, например, изменение пароля пользователя. Класс ViewSet позволяет это сделать. Рассмотрим пример, в котором нужно по определённому адресу выдавать не всю информацию о статье (модели Article), а только её текст. Viewset в этом случае может выглядеть так:

```
...
from rest_framework.decorators import action
...
```

```
class ArticleViewSet(viewsets.ViewSet):
    renderer_classes = [JSONRenderer]

    @action(detail=True, methods=['get'])
    def article_text_only(self, request, pk=None):
        article = get_object_or_404(Article, pk=pk)
        return Response({'article.text': article.text})

    def list(self, request):
        articles = Article.objects.all()
        serializer = ArticleSerializer(articles, many=True)
        return Response(serializer.data)

    def retrieve(self, request, pk=None):
        article = get_object_or_404(Article, pk=pk)
        serializer = ArticleSerializer(article)
        return Response(serializer.data)
```

mainapp/views.py

Кроме реализации методов list и retrieve мы добавили свой метод с именем article_text_only:

```
@action(detail=True, methods=['get'])
def article_text_only(self, request, pk=None):
    article = get_object_or_404(Article, pk=pk)
    return Response({'article.text': article.text})
```

Получаем объект Article и возвращаем пользователю только его текст. Чтобы роутер распознал дополнительное действие, нужен декоратор `@action(detail=True, methods=['get']).`

В нём указываем методы, доступные для этого действия, и detail. Параметр detail показывает, работаем ли мы со всей выборкой или с одним объектом. Для одного объекта в адрес добавляется pk.

В urls.py для регистрации в роутере никаких дополнительных действий не требуется. Роутер сам создаст адрес следующего вида: */viewsets/viewset/1/article_text_only/*.

В этом адресе цифра 1 — это pk статьи Article.

GenericViewSet

Этот класс добавляет методы get_queryset и get_object для работы с данными и используется как основа для построения Viewsets и нескольких View.

ModelViewSet

Мы уже немного работали с ModelViewSet. Этот класс основан на GenericViewSet и позволяет быстро построить API для модели данных. Это полезно в тех случаях, когда нужны почти все REST API-запросы для конкретной модели.

```
class ArticleModelViewSet(viewsets.ModelViewSet):
   queryset = Article.objects.all()
   renderer_classes = [JSONRenderer]
   serializer_class = ArticleSerializer
```

mainapp/views.py

Для базового использования достаточно указать queryset и serializer_class. Для настройки можно переопределить любой метод классов ViewSet и ModelViewSet, а также добавить дополнительные действия.

Custom ViewSet

Один из наиболее удобных видов Viewset — Viewset, собранный из нескольких примесей (mixins). Примеси могут быть взяты из DRF, так и являться своими классами. Рассмотрим пример создания такого Viewset.

```
...
from rest_framework import mixins
...

class ArticleCustomViewSet(mixins.CreateModelMixin, mixins.ListModelMixin,
                           mixins.RetrieveModelMixin, viewsets.GenericViewSet):
   queryset = Article.objects.all()
   serializer_class = ArticleSerializer
   renderer_classes = [JSONRenderer, BrowsableAPIRenderer]
```

mainapp/views.py

Основа — GenericViewSet. К нему добавляются нужные классы примеси. Таким образом, от того, какие примеси добавлены, будет зависеть доступность запросов REST API. В этом примере добавился CreateModelMixen, ListModelMixin, RetrieveModelMixin. Это означает, что нам будут доступны запросы get и post. Появятся возможности создавать новые записи, просматривать список или одну запись. Далее стандартно указывается queryset и serializer_class.

Урлы для этого Viewset генерируются также через Router.

Применение

Чаще используются ModelViewSet и Custom Viewset. ModelViewSet удобен, когда нужно большинство методов для одной модели данных, а Custom Viewset — при необходимости только части методов для API. Класс ViewSet используется реже для нестандартных задач и нескольких типов rest-запросов.

Резюме

1. DRF предоставляет возможность использовать различные виды View для решения как любых типовых, так и нестандартных задач.
2. Views удобно группировать во Viewsets и после этого использовать Router для удобной генерации url-адресов.
3. DRF позволяет не только переопределять обработчики основных запросов, но и добавлять свои обработчики для нестандартных действий.

Filtering

Введение

Добавление фильтрации в REST API — стандартная задача. Часто требуется вывести данные только для конкретного пользователя, либо отфильтровать данные по части имени или дате добавления. DRF предоставляет удобные средства для создания фильтров. Рассмотрим наиболее актуальные варианты.

get_queryset

Views и Viewsets, которые содержат свойство queryset, также имеют метод get_queryset для получения выборки данных динамически. Если этот метод не переопределился, по умолчанию он возвращает свойство queryset. Если переопределим этот метод, сможем пользоваться методом filter у менеджера моделей objects. Рассмотрим это на примере:

```
class ArticleQuerysetFilterViewSet(viewsets.ModelViewSet):
   serializer_class = ArticleSerializer
   renderer_classes = [JSONRenderer, BrowsableAPIRenderer]
   queryset = Article.objects.all()

   def get_queryset(self):
       return Article.objects.filter(name__contains='python')
```

mainapp/views.py

В примере используется ModelViewSet. Но это актуально и для других Views и Viewset, у которых есть свойство queryset.

```
 def get_queryset(self):
       return Article.objects.filter(name__contains='python')
```

После переопределения метода get_queryset берём не все данные, а фильтруем статьи Article по части имени. В нашем случае имя должно содержать слово python.

kwargs

В предыдущем примере ‘python’ содержится в коде. Часто мы хотим предоставить пользователю возможность вводить данные для фильтрации. Это можно сделать двумя основными способами:

- через параметр в url-адресе;
- через параметры запроса.

Когда мы используем параметры в url-адресе, то данные фильтра получаем через kwargs запроса. Рассмотрим это на примере:

```
class ArticleKwargsFilterView(ListAPIView):
   serializer_class = ArticleSerializer

   def get_queryset(self):
       name = self.kwargs['name']
       return Article.objects.filter(name__contains=name)
```

mainapp/views.py

Из self.kwargs берётся по ключу параметр name и затем передаётся в фильтр. Адрес при этом должен содержать строковый параметр name:

```
path('filters/kwargs/<str:name>/', views.ArticleKwargsFilterView.as_view())
```

blog/urls.py

Параметры запроса

Если мы используем параметры запроса для передачи параметров, то специальный адрес делать не нужно:

```
...
filter_router = DefaultRouter()
filter_router.register('param', views.ArticleParamFilterViewSet)
...

urlpatterns = [
```

```
    ...
    path('filters/', include(filter_router.urls)),
    ...
```

blog/urls.py

Например, адрес можно оставить таким, какой создаст Router.

Сам Viewset в этом случае будет выглядеть следующим образом:

```
class ArticleParamFilterViewSet(viewsets.ModelViewSet):
    queryset = Article.objects.all()
    serializer_class = ArticleSerializer

    def get_queryset(self):
        name = self.request.query_params.get('name', '')
        articles = Article.objects.all()
        if name:
            articles = articles.filter(name__contains=name)
        return articles
```

mainapp/views.py

В этом случае значения параметров берутся из self.request.query_params.get, или их может и не быть.

Адрес с параметрами имеет следующий вид: ссылка.

DjangoFilter

Метод get_queryset позволяет добавить в приложении любые типы фильтров. Однако при частом использовании будет много одинакового кода по получению параметров из адреса и передачу их в фильтр.

Часто нужно обрабатывать ситуацию, когда параметр может быть или отсутствовать. Для некоторых фильтров, например, фильтр по дате, не придётся получать дату как строку и приводить её в нужный вид.

Библиотека django-filter и возможность её быстрой интеграции в DRF позволяют быстро добавлять фильтры из нескольких полей разного типа.

Для работы с django-filter можно обратиться к её официальной документации. Важно смотреть раздел интеграции с DRF, так как подключение фильтров к DRF немного отличается от подключения фильтров к Django. Для указания, что мы будем использовать django-filter по умолчанию в settings.py, добавляем следующие настройки DRF:

```
REST_FRAMEWORK = {
```

```
    ...
    'DEFAULT_FILTER_BACKENDS': ['django_filters.rest_framework.DjangoFilterBackend']
```

blog/settings.py

Рассмотрим примеры создания и настройки фильтров. В большинстве случаев нам нужно только указать поля, по которым хотим добавить фильтрацию.

```
class ArticleDjangoFilterViewSet(viewsets.ModelViewSet):
    queryset = Article.objects.all()
    serializer_class = ArticleSerializer
    filterset_fields = ['name', 'user']
```

mainapp/views.py

При добавлении во Vlew или Viewset свойства filterset_fields с указанием списка полей получаем готовый фильтр. В браузере при этом появится кнопка «Фильтр», а сами данные будут передаваться как параметры запроса.

Если, кроме указания полей, необходимо настроить их отображение и тип, то сначала создаём фильтр:

```
from django_filters import rest_framework as filters
from .models import Article


class ArticleFilter(filters.FilterSet):
    name = filters.CharFilter(lookup_expr='contains')

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

mainapp/filters.py

В нём указываем модель, для которой создаётся фильтр, список доступных полей, настройку для каждого поля. name = filters.CharFilter(lookup_expr='contains') в этом примере мы указали, что к фильтру в поле name нужно добавить contains для поиска по части имени, а не по полному совпадению.

Важно! *Фильтры рекомендуется создавать в отдельном файле filters.py внутри приложения.*

После того как создали фильтр, импортируем его в файл, где находиться Viewset и указываем нужный фильтр внутри Vlewset:

```python
from .filters import ArticleFilter


class ArticleCustomDjangoFilterViewSet(viewsets.ModelViewSet):
   queryset = Article.objects.all()
   serializer_class = ArticleSerializer
   filterset_class = ArticleFilter
```

mainpp/views.py

Другие типы фильтров

Существуют и другие типы фильтров. Их описание можно найти в официальной документации. В качестве фильтров можно использовать сторонние библиотеки, а также писать свои.

Применение

Фильтрация становится возможна благодаря переопределению метода get_queryset и передачи параметров во views. Однако переопределение get_queryset удобно только в небольших проектах. В большинстве случаев удобно пользоваться django-filter для быстрого создания сложных настраиваемых фильтров.

Pagination

Введение

Постраничный вывод (Pagination) — важная задача для создания REST API. Если количество данных растёт, становится трудно отдавать их в одном запросе. Возрастает нагрузка на сеть. Поэтому практически для всех наборов данных будем создавать Pagination. В DRF можно настроить постраничный вывод для всего проекта и указывать другие Paginations для каждого View.

Настройка

Для настройки постраничного вывода в settings.py добавим следующий код:

```python
REST_FRAMEWORK = {
...
   'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
   'PAGE_SIZE': 5
...
```

blog/settings.py

В этом примере по умолчанию для всех Views и Viewsets будет выбираться PageNumberPagination. Это постраничный вывод на основе номера страницы. PAGE_SIZE показывает, сколько записей будет выводиться на одной странице.

PageNumberPagination

Такой вид постраничного вывода заключается в выводе определённого количества записей на одной странице. Он удобен, когда на клиенте мы выводим данные по страницам с фиксированной длиной. Например, товары в интернет-магазине.

LimitOffsetPagination

Этот вид постраничного вывода более гибкий, чем PageNumberPagination. Обычно используется для API, в которых клиент может сам решать. Например, сколько данных он хочет получить, и с какой точки отсчитывать это количество. Рассмотрим следующий пример:

```
from rest_framework.pagination import LimitOffsetPagination


class ArticleLimitOffsetPagination(LimitOffsetPagination):
   default_limit = 2


class ArticleLimitOffsetPaginatonViewSet(viewsets.ModelViewSet):
   queryset = Article.objects.all()
   serializer_class = ArticleSerializer
   pagination_class = ArticleLimitOffsetPagination
```

mainapp/views.py

Для начала мы создали свой класс на основе LimitOffsetPagination.

```
class ArticleLimitOffsetPagination(LimitOffsetPagination):
   default_limit = 2
```

default_limit — показывает сколько записей по умолчанию будет выводиться, если этот параметр не будет передан в запросе.

```
pagination_class = ArticleLimitOffsetPagination
```

Затем конкретному Viewset указываем класс для постраничного вывода. Этот класс переопределяет настройку по умолчанию.

Запрос с указанием limit и offset будет выглядеть так: ссылка.

Другие виды постраничного вывода

DRF позволяет писать свои классы для постраничного вывода или использовать сторонние библиотеки.

Итоги

В этом занятии мы рассмотрели большую часть актуальных задач, которые могут быть решены с помощью Views, Filters, Pagination. От конкретной задачи будет зависеть, какой из возможных вариантов будем использоваться. Независимо от задачи удобными вариантами будут:

1. Группировка Views во View sets. Так можно будет использовать Router.
2. Указание Pagination по умолчанию.
3. Использование django-filters.

Глоссарий

View — часть архитектуры DRF, для обработки запросов на одном url-адресе

View set — часть архитектуры DRF, для группировки Views в набор

Filter — часть архитектуры DRF, предназначенная для создания и настройки фильтрации.

Pagination — постраничный вывод.

Используемые источники и дополнительные материалы

1. Views DRF.
2. Generic Views DRF.
3. Viewsets DRF.
4. Filtering DRF.
5. Pagination DRF.

Практическое задание

Создать модели Project и ToDo. Сформировать и настроить API для этих моделей.

В этой самостоятельной работе тренируем умения

1. Создавать Views.

2. Добавлять в API фильтрацию.
3. Добавлять в API постраничный вывод.

Зачем?

Чтобы применять эти умения при разработке REST API.

Последовательность действий

1. Установить размер страницы для всех api 100 записей.
2. Выбрать подходящий класс для постраничного вывода.
3. В проекте доработать API следующим образом:
    - модель User: есть возможность просмотра списка и каждого пользователя в отдельности, можно вносить изменения, нельзя удалять и создавать;
    - модель Project: доступны все варианты запросов; для постраничного вывода установить размер страницы 10 записей; добавить фильтрацию по совпадению части названия проекта;
    - модель ToDo: доступны все варианты запросов; при удалении не удалять ToDo, а выставлять признак, что оно закрыто; добавить фильтрацию по проекту; для постраничного вывода установить размер страницы 20.
4. (Задание со *) В модели ToDo добавить фильтрацию по дате создания. Передадим 2 даты, дату начала и окончания (ссылка).