DRF: права, лимиты, пагинация, фильтрация, поиск | Я.Шпора

Во всех примерах будет использована модель `Cat`:

```
from django import models


class Cat(models.Model):
    name = models.CharField(max_length=16)
    color = models.CharField(max_length=16)
    birth_year = models.IntegerField(blank=True, null=True)

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

Контроль доступа: Permissions

Чтобы определить права доступа на уровне проекта, в словаре настроек `REST_FRAMEWORK` задают параметр `DEFAULT_PERMISSION_CLASSES`.

```
# settings.py
...

REST_FRAMEWORK = {

    ...

    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ],
}
```

На уровне проекта можно установить один из четырёх вариантов доступа:

- `AllowAny` — всё разрешено, любой пользователь (даже аноним) может выполнить любой запрос.
- `IsAuthenticated` — только аутентифицированные пользователи могут получить доступ к API и выполнить любой запрос. Остальным вернётся ответ **"401 Unauthorized".**
- `IsAuthenticatedOrReadOnly` — анонимы могут делать запросы только на чтение; запросы на создание, удаление или редактирование информации доступны только аутентифицированным пользователям.
- `IsAdminUser` — выполнение запросов разрешено только пользователям с правами администратора — тем, у которых свойство `user.is_staff` равно `True`.

Права доступа в DRF можно настроить на уровне представлений.

На примере вьюсета: настройка прав с помощью атрибута `permission_classes`:

```
from rest_framework import permissions

...

class CatViewSet(viewsets.ModelViewSet):
    queryset = Cat.objects.all()
    serializer_class = CatSerializer
    # Устанавливаем разрешение:
    permission_classes = (permissions.IsAuthenticatedOrReadOnly,)
```

У разрешений на уровне проекта приоритет ниже, чем у разрешений на уровне представления.

Собственные разрешения создаются наследованием от класса `BasePermission`. В классе `BasePermission` описаны два метода:

- `has_permission()` определяет разрешения на уровне запроса;
- `has_object_permission()` устанавливает разрешения на уровне объекта.

Доступ будет разрешён, если методы вернут `True`.

```python
from rest_framework import permissions

class OwnerOrReadOnly(permissions.BasePermission):

    def has_permission(self, request, view):
        return (
                request.method in permissions.SAFE_METHODS
                or request.user.is_authenticated
            )

    def has_object_permission(self, request, view, obj):
        return obj.owner == request.user
```

Обратите внимание:

- Метод `has_object_permission` никогда не выполняется для представлений, возвращающих коллекции объектов или создающих новый объект модели (поскольку объект ещё не существует).
- Метод `has_object_permission` вызывается только в том случае, если метод `has_permission` вернул `True`. В противном случае ваш кастомный пермишен сразу же вернёт `False`, не вызывая метод `has_object_permission`.
- По умолчанию оба метода возвращают значение `True`. Поэтому если в кастомном пермишне не переопределить эти методы — пользователям будет предоставлен полный доступ.

Ограничение количества запросов: Throttling

Тротлинг определяет, можно ли разрешить запрос к API. Отличие от пермишенов в том, что тротлинг устанавливает ограничение на лимит запросов и определяет разрешённую частоту обращений к API.

При превышении лимита возвращается статус-код **429 "Too Many Requests".**

Настройка троттлинга похожа на настройку прав доступа: эти настройки можно указать на уровне проекта и на уровне представления.

Ограничения на уровне проекта: в файле *settings.py* в словарь `REST_FRAMEWORK` необходимо добавить параметры `DEFAULT_THROTTLE_CLASSES` и `DEFAULT_THROTTLE_RATES`.

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.UserRateThrottle',
        'rest_framework.throttling.AnonRateThrottle',
    ],
    'DEFAULT_THROTTLE_RATES': {
        'user': '10000/day',  # Лимит для UserRateThrottle.
        'anon': '1000/day',  # Лимит для AnonRateThrottle.
    }
}
```

Ограничения на уровне представления (во view-классах или вьюсетах): в тело класса добавляют атрибут `throttle_classes`.

```
...

from rest_framework.throttling import AnonRateThrottle

...

class CatViewSet(viewsets.ModelViewSet):
    queryset = Cat.objects.all()
    serializer_class = CatSerializer
    throttle_classes = (AnonRateThrottle,)  # Подключили класс Ano
```

Кастомные лимиты

Лимиты `user` и `anon` встроены в DRF и работают «из коробки». Но можно описывать и применять собственные лимиты (их называют «скоуп», от англ. *scope* — «пределы, границы»). Для этого в *settings.py* в настройках DRF нужно **подключить класс** `ScopedRateThrottle`; после этого нужно придумать название для нового скоупа и указать его в `DEFAULT_THROTTLE_RATES`:

```
REST_FRAMEWORK = {
    ...
    'DEFAULT_THROTTLE_CLASSES': [
        ...
```

```
        'rest_framework.throttling.ScopedRateThrottle',
    ],
    'DEFAULT_THROTTLE_RATES': {
        ...
        'user': '10000/day',
        'anon': '1000/day',
        # Имена (ключи) для scope придумывает разработчик,
        # в меру собственной фантазии:
        'low_request': '1/minute',
    }
}
```

Теперь можно применить новый скоуп: его можно подключить к отдельным view-классам или вьюсетам; скоуп указывается в атрибуте `throttle_scope`:

```
class CatViewSet(viewsets.ModelViewSet):
    queryset = Cat.objects.all()
    serializer_class = CatSerializer
    # Для всех пользователей установим кастомный лимит: 1 запрос в
    throttle_scope = 'low_request'
```

Кастомные тротлинг-классы

Собственные тротлинг-классы должны наследоваться от базового класса `BaseThrottle`. В классе-наследнике нужно описать метод `allow_request`. Этот метод должен возвращать `True`, если нужно разрешить запрос, и `False` — если запрос следует отклонить.

```
import datetime
from rest_framework import throttling


class WorkingHoursRateThrottle(throttling.BaseThrottle):

    def allow_request(self, request, view):
        now = datetime.datetime.now().hour
        if now >= 3 and now <= 5:
```

```
            return False
        return True
```

Пагинация

Пагинация в API позволяет выводить информацию частями.

Настройки пагинации можно указать на уровне проекта и на уровне представления.

Настройка пагинации на уровне проекта: добавить ключи `DEFAULT_PAGINATION_CLASS` и `PAGE_SIZE` в словарь настроек `REST_FRAMEWORK`. Именно они отвечают за подключение пагинатора и число объектов в выдаче.

```
...

REST_FRAMEWORK = {
    ...
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNu
    'PAGE_SIZE': 5,
}
```

Пагинация «из коробки» будет работать только для дженериков и вьюсетов. Для view-классов пагинацию настраивают иначе; детали реализации можно подсмотреть в исходном коде классов `mixins.ListModelMixin` и `generics.GenericAPIView`.

Настройка пагинации на уровне представления: в атрибуте `pagination_class` view-класса (*Generics* или *ViewSets*) нужно указать класс пагинатора:

```
from rest_framework.pagination import PageNumberPagination


class CatViewSet(viewsets.ModelViewSet):
    queryset = Cat.objects.all()
    serializer_class = CatSerializer
    pagination_class = PageNumberPagination
```

При включённой пагинации запрос к API можно делать с дополнительным параметром `page`:

```
GET http://127.0.0.1:8000/cats/?page=2
```

Есть более гибкий класс для пагинации: `LimitOffsetPagination`. Этот класс даёт клиенту возможность самостоятельно определять, с какого по счёту объекта (параметр `offset`) и какое количество объектов (параметр `limit`) нужно получить.

```
...
from rest_framework.pagination import LimitOffsetPagination


class CatViewSet(viewsets.ModelViewSet):
    queryset = Cat.objects.all()
    serializer_class = CatSerializer
    # Даже если на уровне проекта установлен PageNumberPagination
    # Для котиков будет работать LimitOffsetPagination
    pagination_class = LimitOffsetPagination
```

При пагинации, настроенной через класс `LimitOffsetPagination`, GET-запрос должен выглядеть примерно так:

```
GET http://127.0.0.1:8000/cats/?limit=2&offset=4
```

Базовый класс пагинатора

В базовом классе пагинаторов `BasePagination` определены два метода:

- `paginate_queryset(self, queryset, request, view=None)`: в него передаётся исходный queryset, а возвращает он итерируемый объект, содержащий только данные запрашиваемой страницы;
- `get_paginated_response(self, data)`: принимает сериализованные данные страницы, возвращает экземпляр Response.

Фильтрация объектов

Для упрощения работы с фильтрацией и поиском в Django REST Framework доступны фильтрующие бэкенды.

Необходимый бэкенд можно подключить на уровне проекта или на уровне представления. Работа с бэкендами на уровне представлений работает более гибко.

Подключаемый бэкенд `DjangoFilterBackend` идёт в составе библиотеки `django-filter`.

Установка библиотеки:

```
pip install django-filter
```

Регистрация приложения `django_filters` в списке приложений `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    'rest_framework',
    'django_filters',
    # Обратите внимание: библиотека называется django-filter,
    # а приложение - django_filters.
]
```

Подключение фильтрующего бэкенда **на уровне представления**:

- импортировать необходимый бэкенд,
- в теле класса:
  - в атрибуте `filter_backends` указать фильтрующий бэкенд,
  - в атрибуте `filterset_fields` указать те поля модели, по которым необходима фильтрация.

```
...
from django_filters.rest_framework import DjangoFilterBackend


class CatViewSet(viewsets.ModelViewSet):
    queryset = Cat.objects.all()
    serializer_class = CatSerializer
```

```
    # Указываем фильтрующий бэкенд DjangoFilterBackend
    # Из библиотеки django-filter
    filter_backends = (DjangoFilterBackend,)
    # Фильтровать будем по полям color и birth_year модели Cat
    filterset_fields = ('color', 'birth_year')
```

Теперь можно сделать, например, GET-запрос для получения всех чёрных котиков:

```
http://127.0.0.1:8000/cats/?color=Black
```

Поиск объектов

Для поиска используется backend `SearchFilter`. Он подключается к нужному вьюсету через атрибут `filter_backends`, а в атрибуте `search_fields` указываются поля модели, по которым разрешён поиск.

Поиск можно вести только по текстовым полям типа `CharField` или `TextField`.

```
...

from rest_framework import filters


class CatViewSet(viewsets.ModelViewSet):
    queryset = Cat.objects.all()
    serializer_class = CatSerializer
    # Добавим в кортеж ещё один бэкенд
    filter_backends = (filters.SearchFilter,)
    search_fields = ('name',)
```

По умолчанию поиск работает по частичным совпадениям без учёта регистра.

Поведение поиска можно настроить, добавив специальные символы к названию поля в `search_fields`:

- '^' — «начинается с …»
- '=' — «полное совпадение»
- '@' — полнотекстовый поиск (поддерживается только для PostgreSQL)

- '$' — регулярное выражение

```
search_fields = ('^name')
```

Теперь в GET-запросе можно указывать не полное имя котика, а только его начало:

```
http://127.0.0.1:8000/cats/?search=Vas
```

Поиск можно проводить и по содержимому полей связанных моделей. Доступные для поиска поля связанной модели указываются через нотацию с двойным подчёркиванием: `ForeignKey текущей модели__имя поля в связанной модели`.

```
class CatViewSet(viewsets.ModelViewSet):
    queryset = Cat.objects.all()
    serializer_class = CatSerializer
    filter_backends = (filters.SearchFilter,)
    # Определим, что значение параметра search должно быть
    # началом искомой строки в поле связанной модели.
    search_fields = ('^achievements__name',)
```

Сортировка объектов

Для сортировки можно подключить встроенный бэкенд `OrderingFilter`; поля для сортировки перечисляются в атрибуте `ordering_fields`.

```
...
from rest_framework import filters


class CatViewSet(viewsets.ModelViewSet):
    queryset = Cat.objects.all()
    serializer_class = CatSerializer
    filter_backends = (filters.OrderingFilter,)
    ordering_fields = ('name', 'birth_year')
```

Теперь при GET-запросе вида

```
http://127.0.0.1:8000/cats/?ordering=name
```

…объекты в выдаче будут отсортированы по именам котиков в алфавитном порядке.

На уровне view-класса или вьюсета можно определить сортировку по умолчанию. Если установлен атрибут `ordering`, то переданное ему значение будет использоваться в качестве поля для сортировки по умолчанию при выдаче.

```
class CatViewSet(viewsets.ModelViewSet):
    ...
    ordering_fields = ('name', 'birth_year')
    ordering = ('birth_year',)
```