

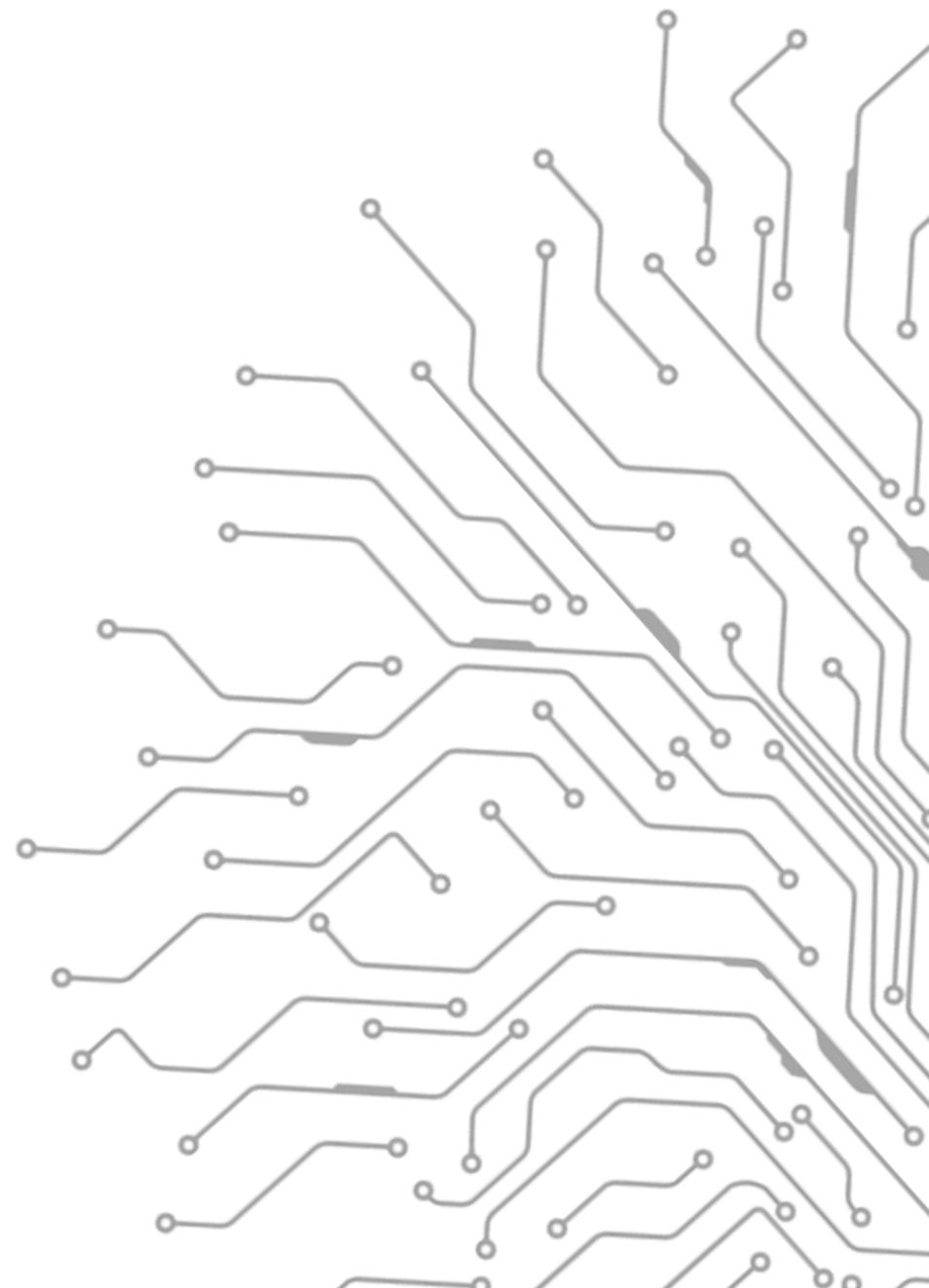
Гайд по Django Rest Framework



Переведено не
профессионально.

Оглавление

- Введение
- Глава 1: CRUD
- Глава 2: Вход и аутентификация
- Глава 3: Пользовательские поля
- Глава 4: Пагинация
- Глава 5: Фильтрация
- Глава 6: Функциональные конечные точки и вложенность API
- Глава 7: Выборочные поля и связанные объекты
- Дополнительно: Попробуйте другой подход

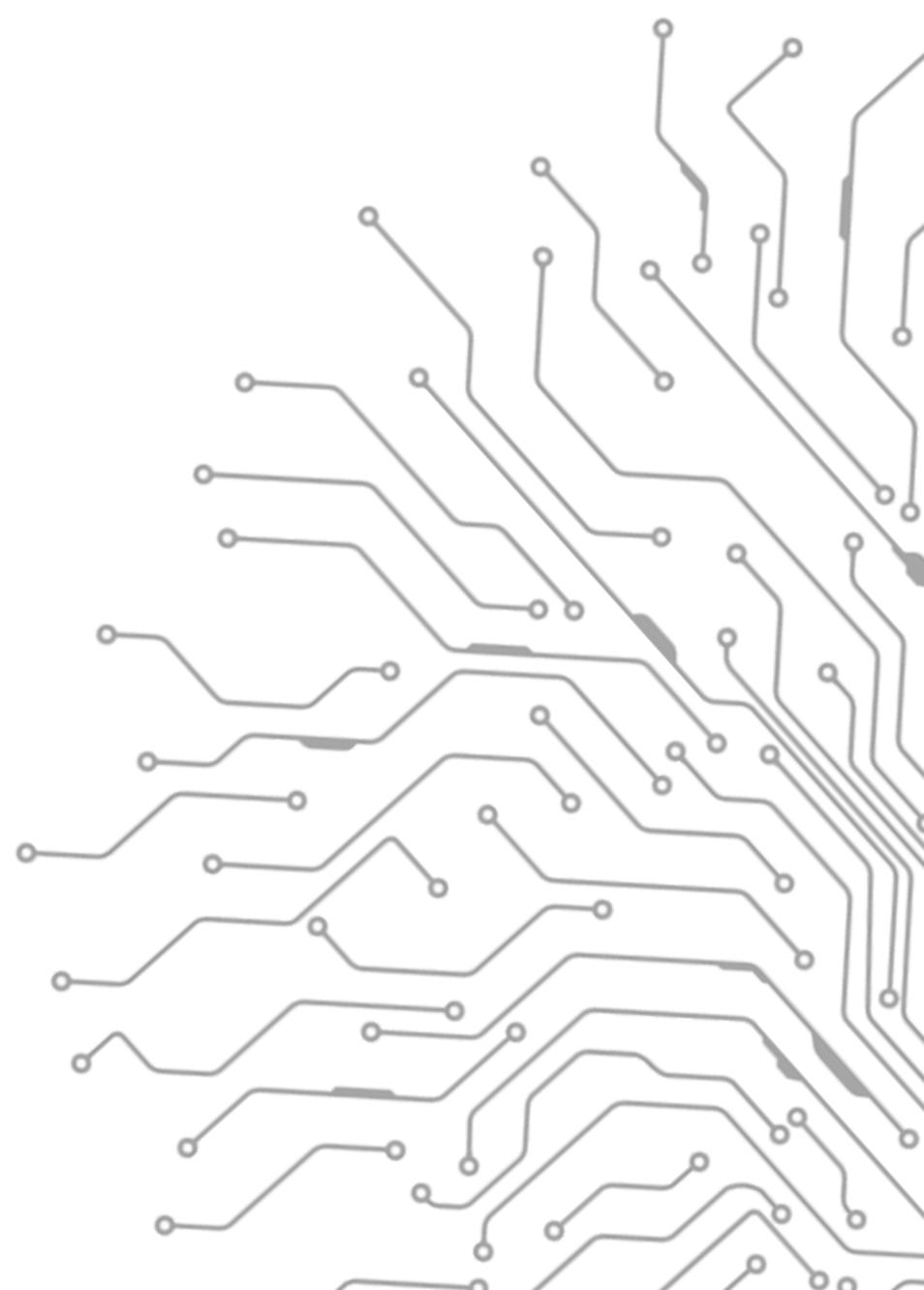


ОБ АВТОРЕ



Доминик увлекается компьютерами всю свою жизнь. Две его страсти - кодинг и преподавание - он программист и преподаватель. Он специализируется в основном на разработке бэкенда и обучении младших разработчиков. Он выбрал работу в **Sunscrapers**, потому что компания глубоко поддерживает сообщество **open-source**. В свободное время Доминик - заядлый геймер.

Dominik Kozaczko
Backend Engineer



От переводчика...

В данном руководстве автор очень хорошо рассматривает все ключевые моменты в **Django Rest Framework**.

Но данное руководство к сожалению не для новичков. Автор уделяет внимание лишь первоначальным моментам. А остальное придется дописывать самому, если читатель внимательный и находчивый то проблем не будет.

В ходе возникновения ошибок или недочетов можете посмотреть код на моем **github**.

Все мои переводы посвящаются сообществу **Python** и **Django**. Эти сообщества помогают новичкам ,так же любым разработчикам у которых возникают ошибки.

Так же хочу выразить огромную благодарность трём замечательным разработчикам:

- Своему ментору (**А.Э**),который **24/7** на связи, поддерживает,и всегда готов помочь.
- Ментору (**Д.А**) опытный разработчик который всегда готов уделить свое время на объяснение сложной темы. Также помогает мне с переводами сложных терминов.
- Ментору (**Ulan**) разработчик который всегда спасает от выгорания ,поддерживает и делиться своим опытом в разработке.

На перевод англоязычных книг меня подтолкнуло отсутствие русской литературы по **Django Rest Framework** ..

В дальнейшем буду переводить остальные хорошие книги по **django**.. Если есть хорошие книги на английском отправляйте в

Telegram:

<https://t.me/kjumabekova>

Всем желаю успехов в разработке.

ВВЕДЕНИЕ

Когда я присоединился к проекту, использующему **Django REST Framework (DRF)**, я часто сталкивался с такими проблемами, как код в стиле "spaghetti" или антипаттерны. Эти проблемы постоянно повторялись, и в какой-то момент я начал задаваться вопросом, откуда берутся проблемы.

Документация **DRF** является исчерпывающей и хорошо организованной, поэтому она должна помочь разработчикам писать более качественный код.

Представим себе разработчика, у которого есть более или менее готовый проект. Теперь они хотят добавить в него **REST API**. Как правило, их первой остановкой будет официальный учебник по **Django REST Framework**.

И тут меня осенило: Учебник по **DRF** написан в обратном порядке. И именно здесь возникает проблема.

Взгляните на него. Вы увидите, что в учебнике сначала показана низкоуровневая универсальность, вместо того чтобы объяснять высокоуровневые аббревиатуры. Когда мы читаем учебник, мы сначала узнаем о деталях представлений, сериализаторов, и только в самом конце - о **ViewSets** - удивительно компактном способе связать все в аккуратное, прозрачное и удобное для управления единое целое.
Но большинство разработчиков так и не доходят до этого момента.

К тому времени они уже имеют относительно функциональный **API** и, в большинстве случаев, решают отказаться от учебника в пользу руководства по **API**, ища способы реализации требований проекта.
Именно поэтому я решил написать собственное руководство по **Django REST Framework**.

В следующих семи главах я проведу вас по **DRF** шаг за шагом, от общего до детального обзора различных его аспектов. Прочтите эту электронную книгу, и вы гарантированно получите понятный и хорошо управляемый код, который не вызовет у вас стыда, когда вы будете показывать или передавать его другим.

Примечание: Вот ссылка на репозиторий, где вы найдете весь код, который я буду рассматривать в этой электронной книге.

ГЛАВА 1: CRUD

Если вы читаете эту статью, то, вероятно, у вас уже есть схема приложения, и теперь вы хотите добавить **REST API** для выполнения основных операций над объектами, таких как **Create, Retrieve, Update и Delete (CRUD)**.

Именно этим мы и займемся в этой главе. Для начала давайте подготовим пример приложения: мы собираемся создать приложение для управления предметами, которые мы одалживаем нашим друзьям.

Нам нужно установить **Django REST Framework**.

```
#Зайдите в вашу папку с проектом, затем активируйте virtualenv создаем приложение rental и устанавливаем djangorestframework
```

```
$ ./manage.py startapp rental  
$ pip install djangorestframework
```

Далее мы изменим параметр **INSTALLED_APPS** в файле **settings.py**.

```
INSTALLED_APPS = [  
    #другие приложения  
    'rental',  
    'rest_framework',  
]
```

rental/models.py

```
from django.db import models  
  
class Friend(models.Model):  
    name = models.CharField(max_length=100)  
  
class Belonging(models.Model):  
    name = models.CharField(max_length=100)  
  
class Borrowed(models.Model):  
    what = models.ForeignKey(Belonging, on_delete=models.CASCADE)  
    to_who = models.ForeignKey(Friend, on_delete=models.CASCADE)  
    when = models.DateTimeField(auto_now_add=True)  
    returned = models.DateTimeField(null=True, blank=True)
```

Чтобы сделать наши объекты доступными через **API**, нам необходимо выполнить сериализацию для текстового отображения данных, содержащихся в объекте. По умолчанию здесь используется формат **JSON**, хотя **DRF** допускает сериализацию в **XML** или **YAML**. Обратный процесс называется десериализацией.

```
from rest_framework import serializers
from . import models

class FriendSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.Friend
        fields = ('id', 'name')

class BelongingSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.Belonging
        fields = ('id', 'name')

class BorrowedSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.Borrowed
        fields = ('id', 'what', 'to_who', 'when', 'returned')
```

Теперь нам нужно создать представления, которые будут обрабатывать операции, которые мы хотим выполнять над нашими объектами.

Рассмотрим их названия и возможные способы сочетания с методами, доступными в стандарте **HTTP**:

Create - этот метод довольно прост. Стандартная поддержка для него исходит от метода **HTTP POST**.

Поскольку мы создаем здесь элемент набора (**ID** объекта будет определен только сейчас), мы рассматриваем этот метод как операцию над списком: создание элемента.

Retrieve - здесь у нас есть два варианта: мы можем загрузить список объектов заданного типа (**list**) или один конкретный объект (**retrieve**). В обоих случаях адекватным методом **HTTP** будет **GET**.

Update - здесь доступны два **HTTP**-метода: **PUT** и **PATCH**. Разница между ними заключается в том, что, согласно определению, **PUT** требует все атрибуты объекта - включая те, которые не изменились.

PATCH, с другой стороны, позволяет ввести только те поля, которые изменились. Именно поэтому он более популярен.

Использование метода **PUT** или **PATCH** для обновления нескольких объектов встречается редко, и **DRF** поддерживает только обновление одного объекта в своем стандартном **CRUD**.

Delete - удаляет один или много объектов. **HTTP**-методом здесь будет **DELETE**. На практике, по причинам безопасности, обычно невозможно удалить несколько объектов одновременно. **DRF** поддерживает эту операцию только для отдельных объектов в своем стандартном **CRUD**. Давайте подытожим все вышесказанное:

Operation	HTTP method	Endpoint type
Create	POST	list
Retrieve many	GET	list
Retrieve one	GET	detail
Update	PUT / PATCH	detail
Delete	DELETE	detail

Для поддержки такого набора операций **DRF** предоставляет удобный инструмент под названием **ViewSet**. Он переносит идею стандартных представлений на основе классов из **Django** на новый уровень. Он упаковывает вышеупомянутый набор в один класс и автоматически создает соответствующие пути **URL**.

Давайте посмотрим, как это работает.

Для начала создадим **ViewSet**, который будет поддерживать наши модели. **DRF** предоставляет **ModelViewSet**, благодаря которому мы можем сократить необходимый объем кода до минимума:

`rental/api_views.py`

```
from rest_framework import viewsets
from . import models
from . import serializers

class FriendViewSet(viewsets.ModelViewSet):
    queryset = models.Friend.objects.all()
    serializer_class = serializers.FriendSerializer

class BelongingViewSet(viewsets.ModelViewSet):
    queryset = models.Belonging.objects.all()
    serializer_class = serializers.BelongingSerializer

class BorrowedViewSet(viewsets.ModelViewSet):
    queryset = models.Borrowed.objects.all()
    serializer_class = serializers.BorrowedSerializer
```

И теперь мы можем перейти к последней части: подключить все это к дереву **URL** нашего проекта.

Мы можем воспользоваться очень удобным инструментом: маршрутизаторами(**routers**). **DRF** предоставляет два наиболее важных класса, которые отличаются только тем, что один из них показывает структуру **API** при загрузке `/ (root)`, а другой - нет.
Наши наборы представлений будут подключены следующим образом:

api.py (глобальный, рядом с settings.py)

```
from rest_framework import routers
from rentall import api_views as rental_views

router = routers.DefaultRouter()
router.register(r'friends', rental_views.FriendViewSet)
router.register(r'belongings', rental_views.BelongingViewSet)
router.register(r'borrowings', rental_views.BorrowedViewSet)
```

urls.py (global)

```
from django.urls import include, path
from django.contrib import admin
from .api import router
urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/', include(router.urls)),
]
```

Давайте проверим созданный нами API.

```
$ ./manage.py makemigrations
$ ./manage.py migrate
$ ./manage.py runserver
```

Откройте этот адрес в браузере (в стандартной конфигурации):
<https://127.0.0.1:8000/api/v1/>.

DRF автоматически создает представления, которые позволяют выполнять API-запросы непосредственно из браузера:

The screenshot shows the Django REST framework's browsable API interface. On the left, under 'Api Root', there is a description: 'The default basic root view for DefaultRouter'. Below it, there is a 'GET /api/v1/' button. To its right, there are 'OPTIONS' and 'GET' buttons. On the right, under 'Borrowed Viewset List', there is a 'GET /api/v1/borrowings/' button. To its right, there are 'OPTIONS' and 'GET' buttons. Below these buttons, there is a JSON response for the 'HTTP 200 OK' status, which includes fields like 'friends', 'belongings', and 'borrowings'. At the bottom, there is a form with fields for 'What', 'To who', and 'Returned', and a 'POST' button. There are also 'Raw data' and 'HTML form' tabs at the bottom.

Экспериментируйте и проверяйте результаты своей работы в панели **django-admin**. Вот так мы получаем **API**, поддерживающий **CRUD** для наших моделей. Обратите внимание, что у нас пока нет никакой защиты от неавторизованного доступа.

В следующей главе я более подробно рассмотрю процесс входа и регистрации пользователей.

Код из этой главы можете найти тут:



<https://github.com/jumabekova06/Guide-to-the-Django-Rest-API>

Глава 2: Вход и аутентификация

В предыдущей главе я показал вам, как подготовить **API**, реализующий базовый **CRUD** на объектах. В этот раз я собираюсь более подробно рассмотреть вход в **API** и регулирование прав доступа.

Среди случаев использования **REST API** можно выделить две доминирующие группы: **(1)** односторонние приложения (**SPA**), использующие возможности браузера, и **(2)** мобильные приложения.

Для первых все, что нам нужно - это стандартный механизм поддержки сессий, предоставляемый **Django** и поддерживаемый **DRF** по умолчанию. К сожалению, мы не можем использовать этот механизм в мобильных приложениях, где гораздо более распространен вход с помощью токена. Вот как это происходит: при запуске приложения мы указываем данные для входа, приложение подключается к **API**, который генерирует токен, который затем сохраняется. Таким образом, пользователям не нужно запоминать логин и пароль - или заставлять устройство запоминать эти данные и подвергать пользователей приложения риску.

DRF предоставляет механизм аутентификации токенов, и вы можете прочитать о нем в официальной документации. Для наших целей описание слишком подробное, но будет полезно взглянуть на него, когда вы закончите читать мое руководство.

Далее мы будем использовать библиотеку `djoser`.

\$ pip install djoser

Давайте начнем с базовой конфигурации:

settings.py/

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

REST_FRAMEWORK = {

'DEFAULT_AUTHENTICATION_CLASSES': (
    'rest_framework.authentication.TokenAuthentication',
    'rest_framework.authentication.SessionAuthentication',
),
}
```

Мы также можем добавить токен входа в наши URL-адреса:

urls.py (global)

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/', include(router.urls)),
    path('api/auth/', include('djoser.urls.authtoken')),
]
```

В заключение мы завершаем миграцию, связанную с токенами:

```
$ ./manage.py migrate
```

С этого момента мы можем получить токен входа в систему с помощью **REST API**:

(Зарегистрированный пользователь должен быть)

```
$ curl -X POST -d '{"username": "admin", "password": "top_secrets"}' -H  
'Content-Type: application/json' localhost:8000/api/auth/token/login/
```

В ответ вы получите что-то вроде этого:

```
{"auth_token": "fe9a080cf91acb8ed1891e6548f2ace3c66a109f"}
```

Давайте теперь защитим наши представления от неавторизованного входа. **Django REST Framework** предлагает несколько классов разрешений, которые мы можем использовать для защиты нашего **API** от неавторизованного входа.

По умолчанию используется разрешение **'rest_framework.permissions.AllowAny'**, которое - как следует из названия - позволяет всем делать все.

Давайте защитим **API** так, чтобы доступ был разрешен только зарегистрированным пользователям.

Для этого нам нужно изменить файл **settings.py**, добавив в него следующую запись:

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

Давайте проверим, защищен ли наш **API**:

```
$ curl -X GET http://127.0.0.1:8000/api/v1/friends/  
{"detail": "Authentication credentials were not provided."}
```

(наш запрос)

(ответ)

Работает! Без токена мы получили только ошибку. Давайте воспользуемся токеном, который мы получили ранее.

```
curl -X GET http://127.0.0.1:8000/api/v1/friends/ -H 'Authorization: Token  
fe9a080cf91acb8ed1891e6548f2ace3c66a109f'  
[{"id": 1, "name": "John Doe"}]
```

ПРИМЕЧАНИЕ: В целях безопасности очень важно, чтобы рабочий **API** был доступен только через **https**.

Однако установка класса по умолчанию, который управляет разрешениями для всего **API**, не является единственным вариантом. Мы также можем установить различные способы обработки разрешений отдельно для каждого набора **ViewSet**, задав атрибут **permission_classes**:

```
class MyViewSet(viewsets.ModelViewSet):
    permission_classes = [permissions.DjangoModelPermissions]
```

Просмотрите документацию **DRF**, чтобы узнать больше о классах разрешений по умолчанию.

Определение разрешений(**permissions**) основывается на анализе запроса и возвращает значение **bool (True / False)**.

Давайте продолжим пример, который я показал вам в предыдущей главе: приложение, помогающее управлять предметами, которые мы одалживаем нашим друзьям.

Что произойдет, если наши друзья заинтересуются нашим приложением и захотят им воспользоваться? Чтобы справиться с этим, нам нужно включить регистрацию пользователей. Но сначала мы должны подготовить разрешения, чтобы только владельцы предметов могли изменять указанные предметы.

Регистрацию пользователей можно осуществить разными способами, и я опущу этот этап, чтобы вы могли разобраться в нем самостоятельно.

Подсказка: хорошей идеей будет использование библиотек **djoser** или **rest_auth**.

Давайте добавим в наши модели информацию о владельцах элементов. Самый удобный способ - создание миксина или абстрактной модели:

```
class OwnedModel(models.Model):
    owner = models.ForeignKey(
        settings.AUTH_USER_MODEL, on_delete=models.CASCADE, null=True
    )
    class Meta:
        abstract = True
```

```
class Belonging(OwnedModel):
    name = models.CharField(max_length=100)
```

Не забудьте после этого выполнить миграцию базы данных:

```
$ ./manage.py makemigrations
$ ./manage.py migrate
```

Пришло время подготовить класс для проверки прав доступа.

Для его реализации мы можем использовать один из следующих методов (или оба): '**has_permission**' и '**has_object_permission**'.

Общие права доступа проверяются всегда, а объект проверяется только после того, как общие права доступа приняты. Эти методы должны возвращать **True**, если разрешение было получено, и **False**, если нет. Значение по умолчанию, возвращаемое обоими методами, - **True**. Если мы используем в представлении несколько классов проверки прав доступа, все они должны успешно пройти тест (результаты будут объединены с помощью **AND**).

Наше представление будет проверять права доступа для элемента, поэтому нам нужно реализовать права доступа следующим образом:

```
from rest_framework import permissions

class IsOwner(permissions.BasePermission):
    message = 'Not an owner.'

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

Атрибут **message** позволяет задать индивидуальное сообщение об ошибке, когда право доступа не предоставлено пользователю.

Если мы хотим разрешить всем пользователям видеть содержимое нашего приложения, то класс будет выглядеть следующим образом:

```
from rest_framework import permissions

class IsOwner(permissions.BasePermission):
    message = 'Not an owner.'

    def has_object_permission(self, request, view, obj):
        if request.method in permissions.SAFE_METHODS:
            return True
        return request.user == obj.owner
```

permissions.SAFE_METHODS содержит список методов **HTTP**, которые не допускают записи, т.е. **GET**, **OPTION** и **HEAD**.

Следующим шагом является запоминание авторизованного пользователя как владельца вновь созданного ресурса. Здесь **DRF** предоставляет нам полезный метод.

Примечание: Этот метод рассматривается в документации в рамках совершенно другой темы.

serializers.py

```
class FriendSerializer(serializers.ModelSerializer):
    owner = serializers.HiddenField(
        default=serializers.CurrentUserDefault()
    )

    class Meta:
        model = models.Friend
        fields = ('id', 'name')
```

В завершение давайте используем наш новый класс **permissions** в ViewSets:

```
from .permissions import IsOwner

class FriendViewSet(viewsets.ModelViewSet):
    queryset = models.Friend.objects.all()
    serializer_class = serializers.FriendSerializer
    permission_classes = [IsOwner]
```

Обратите внимание, что при создании объекта проверяется только право доступа '**has_permission**', поэтому вы можете захотеть ограничить его только авторизованными пользователями. Для этого импортируйте '**rest_framework.permissions.IsAuthenticated**' и добавьте его в атрибут '**permission_classes**'. Помните - оба параметра должны возвращать **True**, чтобы право доступа было предоставлено.

DRF предоставляет нам основные классы прав доступа, и их названия говорят сами за себя:

- **AllowAny**,
- **IsAuthenticated**,
- **IsAuthenticatedOrReadOnly**,
- **IsAdminUser** (the `is_staff` attribute is checked here, not `is_superuser!`),
- **DjangoModelPermissions**,
- **DjangoModelPermissionsOrAnonReadOnly**,
- **DjangoObjectPermissions**.

Последние три основаны на **Django**, и я считаю их особенно интересными. В то время как первые два реализуют стандартные разрешения модели, последний требует использования библиотеки типа **django-guardian**. Подробнее об этом вы можете прочитать в документации **DRF**.

Вы можете использовать и другие библиотеки, но я предпочитаю разрешения **DRY REST**, потому что они следуют принципу "толстых моделей", рекомендованному создателями **Django**.

В следующей главе я расскажу о дополнительной информации в сериализаторах - в частности, в динамически генерируемых полях.

Код из этой главы можете найти тут:



<https://github.com/jumabekova06/Guide-to-the-Django-Rest-API>

Глава 3: Пользовательские поля

В этой главе я покажу вам, как добавить динамические поля в нашу модель сериализатора.

Динамические поля - это поля, которые не являются частью нашей базы данных, и их значение вычисляется на регулярной основе.

Например, я использую замечательную библиотеку **pendulum** для службы **datetime**.

Предположим, что мы хотим отобразить информацию о том, что друг должен вернуть нам одолженную вещь, прямо рядом с его именем.

Вот как будет выглядеть загрузка этой информации в нашу модель данных. Допустим, я хочу узнать, хранит ли друг мои вещи более двух месяцев.

```
friend = Friend.objects.get(id=1)
friend.borrowed_set.filter(returned_isnull=True,
when=pendulum.now().subtract(months=2)).exists()
```

Этот фрагмент кода отлично работает для одного друга. Если бы мы захотели отобразить эти данные для группы людей, мы неизбежно заполнили бы базу данных сотнями (или даже тысячами) запросов. И, как сказал бы Раймонд Хеттингер: Должен быть лучший способ!

И, к счастью, он есть. Мы можем использовать здесь механизм аннотаций. Следующий запрос добавит поле '**ann_overdue**' ко всем элементам **queryset**.

Примечание: Через минуту вы увидите, почему я не использую '**has_overdue**'.

```
Friend.objects.annotate(
    ann_overdue=models.Case(
        models.When(borrowed_returned_isnull=True,
                    borrowed_when_lte=pendulum.now().subtract(months=2),
                    then=True),
        default=models.Value(False),
        output_field=models.BooleanField()
    )
)
```

Давайте проанализируем это подробно. Я использую функцию **Case**, которая принимает в качестве параметра любое количество функций **When** - в нашем случае достаточно одной. Внутри **When** я ввожу условие - если любой из объектов **Borrowed**, связанных с этим другом, имеет пустое (**NULL**) поле "возвращено", а также его поле "когда" содержит дату не менее "двух месяцев назад", то значение будет **True**. Значение по умолчанию - **False**, и результат должен быть отображен в поле **BooleanField**.

Все просто, правда? ;)

Вы можете узнать больше об обусловленных выражениях в наборах запросов(**querysets**) в документации **Django**.

Теперь возникает другой вопрос: куда мы поместим этот код?

Обычно, если мы следуем концепции "толстых моделей", мы должны поместить всю логику внутри модели. Однако это не всегда возможно. Наши модели (или, по крайней мере, их часть) могут быть получены из сторонних приложений. В нашем случае мы имеем полный контроль над моделями, поэтому я пойду по этому пути. Я буду указывать, где нам нужно что-то изменить, если мы имеем ввиду другую ситуацию:

/models.py

```
class FriendQuerySet(models.QuerySet):
    def with_overdue(self):
        return self.annotate(
            ann_overdue=models.Case(
                models.When(borrowed__when_lte=pendulum.now().subtract(months=2),
                            then=True),
                default=models.Value(False),
                output_field=models.BooleanField()
            )
        )
class Friend(OwnedModel):
    name = models.CharField(max_length=100)
    objects = FriendQuerySet.as_manager()

    @property
    def has_overdue(self):
        if hasattr(self, 'ann_overdue'): # in case we deal with annotated object
            return self.ann_overdue
        return self.borrowed_set.filter( # 1
            returned_isnull=True, when=pendulum.now().subtract(months=2)).exists()
```

/api_views.py

```
class FriendViewSet(viewsets.ModelViewSet):
    queryset = models.Friend.objects.with_overdue()
    serializer_class = serializers.FriendSerializer
    permission_classes = [IsOwner]
```

/serializers.py

```
class FriendSerializer(serializers.ModelSerializer):
    owner = serializers.HiddenField(default=serializers.CurrentUserDefault())

    class Meta:
        model = models.Friend
        fields = ('id', 'name', 'has_overdue')
```

Если мы не можем изменять модели, то для представления нужно будет переписать метод 'get_queryset', а в `serializers.py` - логику 'has_overdue':

/api_views.py

```
class FriendViewSet(viewsets.ModelViewSet):
    queryset = models.Friend.objects.with_overdue()
    serializer_class = serializers.FriendSerializer
    def get_queryset(self):
        return super().get_queryset().annotate(
            ann_overdue=models.Case(
                models.When(borrowed_when_lte=pendulum.now().subtract(months=2),
                            then=True),
                default=models.Value(False),
                output_field=models.BooleanField()
            )
        )
```

/serializers.py

```
class FriendSerializer(serializers.ModelSerializer):
    owner = serializers.HiddenField(
        default=serializers.CurrentUserDefault())
    has_overdue = serializers.SerializerMethodField()
    class Meta:
        model = models.Friend
        fields = ('id', 'name', 'owner', 'has_overdue')

    def get_has_overdue(self, obj):
        if hasattr(obj, 'ann_overdue'):
            return obj.ann_overdue
        return obj.borrowed_set.filter(returned_isnull=True,
                                       when=pendulum.now().subtract(months=2)).exists()
```

Как вы видите, я определяю поле (или свойство) 'has_overdue', помещаю его в атрибут `Meta.fields` сериализатора и, если требуется, объясняю DRF, как получить значение.

Обратите внимание, как я проверяю наличие атрибута 'ann_overdue'. Таким образом, я получаю очень универсальный код - если аннотация была использована, значение уже вычислено, и мы можем использовать его повторно. Если нет - что ж, нам нужно сделать тяжелую работу самостоятельно.

Как это изменит количество посещений базы данных? Я подготовил примерный набор данных, содержащий **1000** друзей и **10000** вещей (предметов), распределенных случайным образом в течение шестимесячного периода. Затем я провел тест (для наглядности я удалил результаты печати):

/Примечание: я использую `ipython`, чтобы улучшить мою оболочку

```
./manage.py shell
```

```
In [1]: from django.db import connection
In [2]: from rentall.models import Friend
In [3]: for f in Friend.objects.all():
...:     print(f.has_overdue)
...:
In [4]: len(connection.queries)
Out[4]: 1000
# another try using with_overdue method
In [1]: from django.db import connection
In [2]: from core.models import Friend
In [3]: for f in Friend.objects.with_overdue():
...:     print(f.has_overdue)
...:
In [4]: len(connection.queries)
Out[4]: 1
```

Вот и все. Для "толстых моделей" вам просто нужно поместить имя метода или свойства в атрибут **Meta.fields** сериализатора. Дополнительным преимуществом этого является то, что менеджер и свойство '**has_overdue**' также могут быть использованы в панели администратора, так что вы получаете два серьезных улучшения в одном тонком пакете.

Если ваши модели получены из стороннего приложения, вам следует поместить необходимую логику в сериализатор, поскольку она может быть использована несколькими представлениями. К сожалению, в нашем случае логика должна оперировать (аннотировать) набором запросов, поэтому она размещается внутри **View**. Но если бы ее нужно было повторно использовать в других представлениях, мы могли бы сделать миксин только для метода '**get_queryset**'.

Также не забывайте всегда предвидеть воздействие на базу данных и использовать '**select_related**' и/или '**prefetch_related**' при получении **queryset**.

В следующей главе я более подробно рассмотрю пагинацию.

#от переводчика:

могут возникнуть проблемы с **ipython**:
решение - <https://github.com/ipython/ipython/issues/13554#issuecomment-1054087137>

если где-то с копированием кода проблемы то код на **github**:
<https://github.com/jumabekova06/Guide-to-the-Django-Rest-API>

откатите по коммитам, коммиты именно по главам .

ГЛАВА 4: Пагинация

В этой главе я рассмотрю еще одну интересную тему: пагинация.

Почему именно пагинация?

Взгляните на стандартный ответ на одной из наших конечных точек.

```
$ curl http://127.0.0.1:8000/api/v1/friends/  
[{"id":1, "name": "John Doe", "has_overdue":true}, {"id":2, "name": "Frank Tester",  
"has_overdue":false}].
```

Мы получили стандартный список элементов. Пока все хорошо.

Но что, если наша конечная точка возвращает тысячи элементов?

Сериализация и передача такого объема данных может занять достаточно много времени, чтобы пользователь заметил отставание приложения.

Мы можем решить эту проблему с помощью пагинации - разделения результатов на страницы фиксированного размера. Лучшей стратегией здесь является использование метода 'limit + offset', где параметр 'limit', передаваемый в GET, определяет количество элементов на странице, а 'offset' - смещение относительно начала списка.

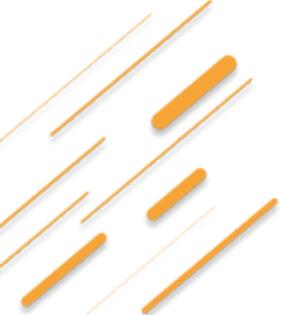
Это универсальный способ обработки более традиционного перехода между подстраницами, а также иногда желаемого метода, называемого "бесконечной прокруткой".

В документации рекомендуется добавить следующие записи в параметры REST_FRAMEWORK в settings.py:

```
REST_FRAMEWORK = {  
    ...  
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.LimitOffsetPagination',  
    'PAGE_SIZE': 100,  
    ...  
}
```

В результате получается следующее (я использовал параметр limit и отформатировал результат для удобочитаемости с помощью json_pp):

```
→ ~ curl http://127.0.0.1:8000/api/v1/friends/?limit=1 | json_pp  
% Total    % Received % Xferd  Average Speed   Time      Time      Time  Current  
          Dload  Upload Total   Spent    Left Speed  
100     98  100     98     0       0  10888      0 --:--:-- --:--:-- --:--:-- 10888  
{  
  "count" : 1,  
  "next" : null,  
  "previous" : null,  
  "results" : [  
    {  
      "has_overdue" : false,  
      "id" : 1,  
      "name" : "John Doe"  
    }  
  ]  
}
```



Как вы можете видеть, результаты были конвертированы. Такая практика оправдана, когда клиент не может обработать **HTTP**-заголовки, но сегодня она постепенно устаревает.

Последние руководства по лучшей практике рекомендуют передавать метаданные в заголовках, разрешая конвертирование по требованию. Ниже мы реализуем это решение.

Для этого мы будем следовать следующим предположениям:

- Конечная точка должна возвращать список элементов в той же структуре, что и первоначально.
- Пагинация осуществляется с помощью параметров '**limit**' и '**offset**'.
- Дополнительные метаданные включаются в соответствующие заголовки.
- Код должен использовать конвертирование по требованию (обеспечивается параметром).
- Код всегда содержит ссылки в заголовках - даже если было выбрано конвертирование.

Примечание: Уже существует библиотека **django-rest-framework-link-header-pagination**, но она не реализует механизм ограничения/смещения, который нас здесь интересует.

Самым простым решением будет наследование класса **rest_framework.pagination.LimitOffsetPagination** поскольку большая часть логики реализована в нем.

Для начала давайте разберемся с параметром, который включает конвертирование:

/pagination.py (в той папке в которой находится **settings.py**)

```
from collections import OrderedDict
from rest_framework.pagination import LimitOffsetPagination
from rest_framework.response import Response
from rest_framework.utils.urls import replace_query_param, remove_query_param

class HeaderLimitOffsetPagination(LimitOffsetPagination):
    def paginate_queryset(self, queryset, request, view=None):
        self.use_envelope = False
        if str(request.GET.get("envelope")).lower() in ["true", "1"]:
            self.use_envelope = True
        return super().paginate_queryset(queryset, request, view)
```

Позже мы можем написать метод, который возвращает данные:

```
def get_paginated_response(self, data):
    next_url = self.get_next_link()
    previous_url = self.get_previous_link()
    links = []
```

```

links = []
for label, url in (
    ("first", first_url),
    ("next", next_url),
    ("previous", previous_url),
    ("last", last_url),
):
    if url is not None:
        links.append('<{}>; rel="{}"'.format(url, label))

headers = {"Link": ", ".join(links)} if links else {}

if self.use_envelope:
    return Response(
        OrderedDict(
            [
                ("count", self.count),
                ("first", first_url),
                ("next", next_url),
                ("previous", previous_url),
                ("last", last_url),
                ("results", data),
            ]
        ),
        headers=headers,
    )
return Response(data, headers=headers)

```

#Этот кусочек кода лучше взять на [github](#):

<https://github.com/jumabekova06/Guide-to-the-Django-Rest-API/blob/master/core/pagination.py>

Чтобы все это работало в соответствии с лучшими методиками, нам нужны ссылки только на первую и последнюю страницу.

Давайте добавим эти два метода:

```

def get_first_link(self):
    if self.offset <= 0:
        return None
    url = self.request.build_absolute_uri()
    return remove_query_param(url, self.offset_query_param)

def get_last_link(self):
    if self.offset + self.limit >= self.count:
        return None
    url = self.request.build_absolute_uri()
    url = replace_query_param(url, self.limit_query_param, self.limit)
    offset = self.count - self.limit
    return replace_query_param(url, self.offset_query_param, offset)

```

Осталось только дописать метод 'get_paginated_response' следующим образом:

```

def get_paginated_response(self, data):
    next_url = self.get_next_link()
    previous_url = self.get_previous_link()
    first_url = self.get_first_link()
    last_url = self.get_last_link()

    links = []
    for label, url in (
        ("first", first_url),
        ("next", next_url),
        ("previous", previous_url),
        ("last", last_url),):

```

```

    if url is not None:
        links.append('<{}>; rel="{}"'.format(url, label))

headers = {"Link": " ".join(links)} if links else {}

if self.use_envelope:
    return Response(
        OrderedDict(
            [
                ("count", self.count),
                ("first", first_url),
                ("next", next_url),
                ("previous", previous_url),
                ("last", last_url),
                ("results", data),
            ]
        ),
        headers=headers,
    )
return Response(data, headers=headers)

```

Куда поместить весь этот код?

Лучше всего поместить этот код в отдельный файл, который можно легко импортировать из любой точки проекта.

Предположим, что мы создадим файл '**pagination.py**', содержащий описанный выше класс, в нашем приложении для аренды книг.

Мы изменим конфигурацию **REST_FRAMEWORK** следующим образом:

```

REST_FRAMEWORK = {
    ...
    'DEFAULT_PAGINATION_CLASS': 'core.pagination.HeaderLimitOffsetPagination',
    'PAGE_SIZE': 100,
}

```

Вы также можете использовать библиотеку, которую я подготовил вместе с кодом выше, установив '**pip install hedju**', а затем в качестве **DEFAULT_PAGINATION_CLASS** вы можете использовать '**hedju.HeaderLimitOffsetPagination**'.

Поскольку все готово, осталось только протестировать **API**; **curl** с параметром **-v** покажет нам заголовки (я удалил неактуальную информацию):

```
$ curl "http://127.0.0.1:8000/api/v1/friends/?limit=1" -v
```

```

* Connection #0 to host 127.0.0.1 left intact
[{"id":1,"name":"Karygach","has_overdue":false}]%
→ ~ curl "http://127.0.0.1:8000/api/v1/friends/?limit=1&envelope=true"
[1] 163327
zsh: no matches found: "http://127.0.0.1:8000/api/v1/friends/?limit=1"
[1] + 163327 exit 1      curl "http://127.0.0.1:8000/api/v1/friends/?limit=1"
→ ~

```

Готово!

В качестве бонуса я хотел бы упомянуть о поддержке навигации по заголовкам в библиотеке **requests**:

In [1]: import requests

In [2]: result = requests.get('http://127.0.0.1:8000/api/v1/friends/?limit=1')

In [3]: result.links

Out[3]:

```
{'next': {'url': 'http://127.0.0.1:8000/api/v1/friends/?limit=1&offset=1',
'rel': 'next'},
'last': {'url': 'http://127.0.0.1:8000/api/v1/friends/?limit=1&offset=1',
'rel': 'last'}}
```

Вот и все, друзья! В следующей главе я собираюсь обсудить тему фильтрации списка данных.



ГЛАВА 5: Фильтрация

Готовы начать работу над фильтрацией? Давайте приступим!

В предыдущей главе мы ограничили объем одновременно загружаемых данных с помощью пагинации. В этот раз давайте подумаем о том, как мы можем легко фильтровать и искать наши ресурсы.

Конечная точка списка арендованных предметов (`/api/v1/borrowed/`) отображает все предметы, независимо от того, были они возвращены или нет.

Имеет смысл отфильтровать этот список по полю "возвращено". Таким образом мы можем проверить, какие элементы еще не были возвращены. Параметр, задающий такую фильтрацию, будет передан через GET, например.

```
$ curl http://127.0.0.1:8000/api/v1/borrowings/?missing=true
```

Мы можем относительно просто решить эту задачу в методе ``get_queryset``.

```
class BorrowedViewSet(viewsets.ModelViewSet):
    queryset = models.Borrowed.objects.all()
    serializer_class = serializers.BorrowedSerializer
    permission_classes = [IsOwner]
    def get_queryset(self):
        qs = super().get_queryset()
        only_missing = str(self.request.query_params.get('missing')).lower()
        if only_missing in ['true', '1']:
            return qs.filter(returned_isnull=True)
        return qs
```

BTW. Напомню, что поле `'returned'` - это поле даты. Если оно содержит **NULL**, это означает, что элемент еще не был возвращен. Именно поэтому здесь используется фильтрация.

Такой реализации достаточно для простых случаев использования. Но при большем количестве переменных, которые мы можем захотеть отфильтровать, это может быстро превратиться в беспорядок, которым трудно управлять. Должен быть лучший способ обработки верно?

К счастью, он есть. **Django REST Framework** позволяет разработчикам использовать библиотеку **django-filter**, которая значительно упрощает определение и управление фильтрами.

Сначала нам нужно установить библиотеку с помощью **pip**:

```
$ pip install django-filter
```

Затем мы обновляем наши настройки:

```
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS': ('django_filters.rest_framework.DjangoFilterBackend',)
    ....}
```

Самый простой способ решить эту задачу - добавить поля, по которым мы хотим фильтровать, в атрибут **'filterset_fields'** в соответствующем представлении, например.

```
class BorrowedViewSet(viewsets.ModelViewSet):  
    queryset = models.Borrowed.objects.all()  
    serializer_class = serializers.BorrowedSerializer  
    permission_classes = [IsOwner]  
    filterset_fields = ('to_who', ) # здесь идет фильтрация
```

Это позволяет нам вести учет по лицам, одолжившим у нас предмет.

```
$ curl http://127.0.0.1:8000/api/v1/borrowings/?to_who=1
```

```
[{"id":1,"what":1,"to_who":1,"when":"2018-01-01T12:00:00Z","returned":null},  
 {"id":3,"what":1,"to_who":1,"when":"2019-04-17T06:35:22.000236Z","returned":null},  
 {"id":4,"what":1,"to_who":1,"when":"2019-04-17T06:35:  
 36.546848Z","returned":null}]
```

К сожалению, у этого метода есть одно очень серьезное ограничение: мы можем задавать только конкретные значения, за исключением **NULL**.

Частичным решением этой проблемы является замена **filterset_fields** на словарь. Ключами являются имена полей, а значением - список допустимых вложенных фильтров, совместимых с нотацией **Django**, например:

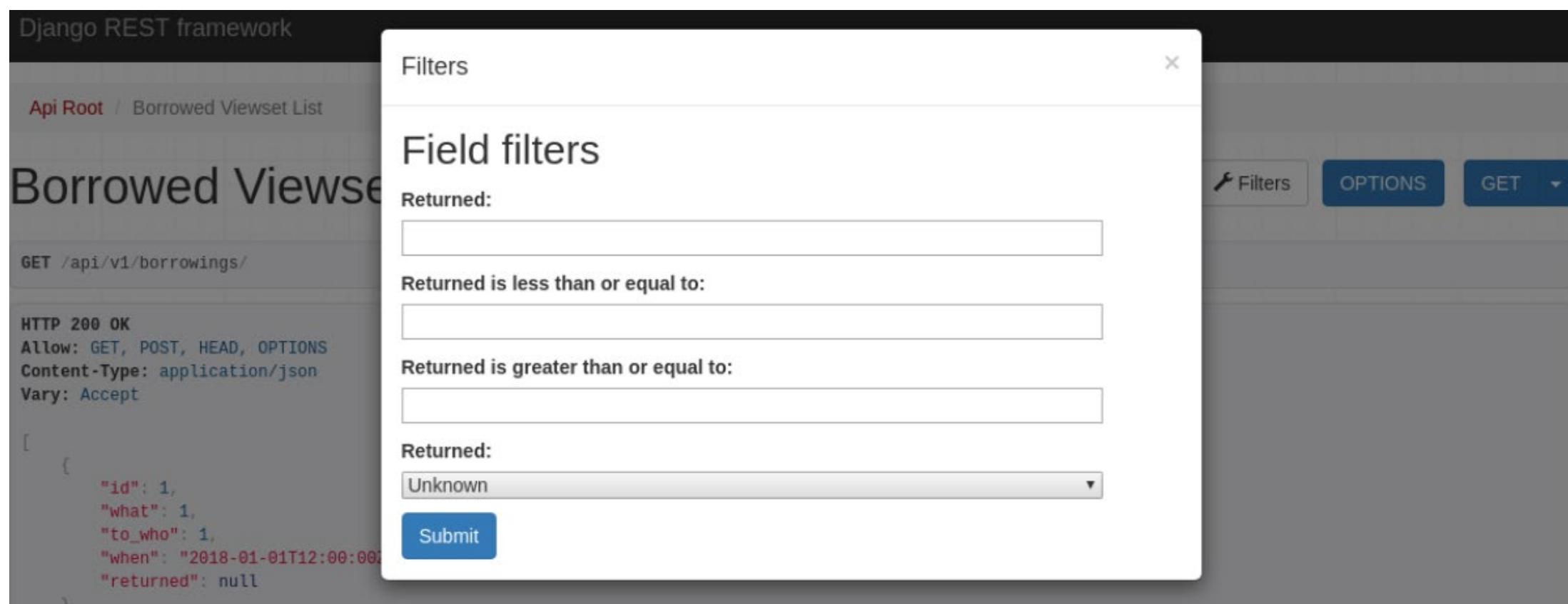
```
class BorrowedViewSet(viewsets.ModelViewSet):  
    queryset = models.Borrowed.objects.all()  
    serializer_class = serializers.BorrowedSerializer  
    permission_classes = [IsOwner]  
    filterset_fields = {  
        'returned': ['exact', 'lte', 'gte', 'isnull']  
    }
```

Это позволяет фильтровать список предметов аренды по дате возврата, включая некоторые полезные подфильтры, такие как **'lte'** и **'gte'**, а также **'isnull'** для отображения запаздывания:

```
$ curl http://127.0.0.1:8000/api/v1/borrowings/?returned_isnull=True
```

```
[{"id":1,"what":1,"to_who":1,"when":"2018-01-01T12:00:00Z","returned":null},  
 {"id":2,"what":2,"to_who":2,"when":"2019-04-16T18:46:16.646649Z","returned":"2019-04-  
 16T18:46:13Z"}, {"id":3,"what":1,"to_  
 who":1,"when":"2019-04-17T06:35:22.000236Z","returned":null},  
 {"id":4,"what":1,"to_who":1,"when":"2019-04-17T06:35:  
 36.546848Z","returned":null}]
```

Более того, это решение позволяет разработчикам отображать параметры фильтрации при просмотре **API** в режиме **HTML** (обратите внимание, что подсказки иногда могут сбивать с толку).



Написание собственного определения **FilterSet**

Все, что я описал выше, ничто по сравнению с тем, чего мы можем достичь, написав собственное определение **FilterSet**.

Давайте начнем с простого примера.

Чтобы увидеть, как это работает, мы реализуем предыдущую функциональность: поиск невозвращенных элементов.

Но теперь мы сделаем это таким образом, чтобы не раскрывать нотацию **Django**.

Для этого нам нужно использовать поле **BooleanFilter**, которое будет разбирать переданное значение. В его параметрах мы определяем поле, которое хотим просмотреть, и конкретное значение выражения, которому будет передано его значение:

```
class BorrowedFilterSet(django_filters.FilterSet):
    missing = django_filters.BooleanFilter(field_name='returned', lookup_expr='isnull')
    class Meta:
        model = models.Borrowed
        fields = ['what', 'to_who', 'missing']
```

Затем мы передаем наш **BorrowedFilterSet** в **ViewSet**:

```
class BorrowedViewSet(viewsets.ModelViewSet):
    queryset = models.Borrowed.objects.all()
    serializer_class = serializers.BorrowedSerializer
    permission_classes = [IsOwner]
    filterset_class = BorrowedFilterSet # here
```

Теперь мы можем сделать следующее:

```
$ curl http://127.0.0.1:8000/api/v1/borrowings/?missing=True
```

Результат должен быть таким же, как и раньше.

Создание поля, позволяющего фильтровать устаревшие предметы аренды, - это лишь небольшая дополнительная работа. Мы также создадим поле **BooleanFilter**, но на этот раз передадим ему имя метода (он также может быть вызываемым), который будет выполнять фильтрацию на переданном **QuerySet**.

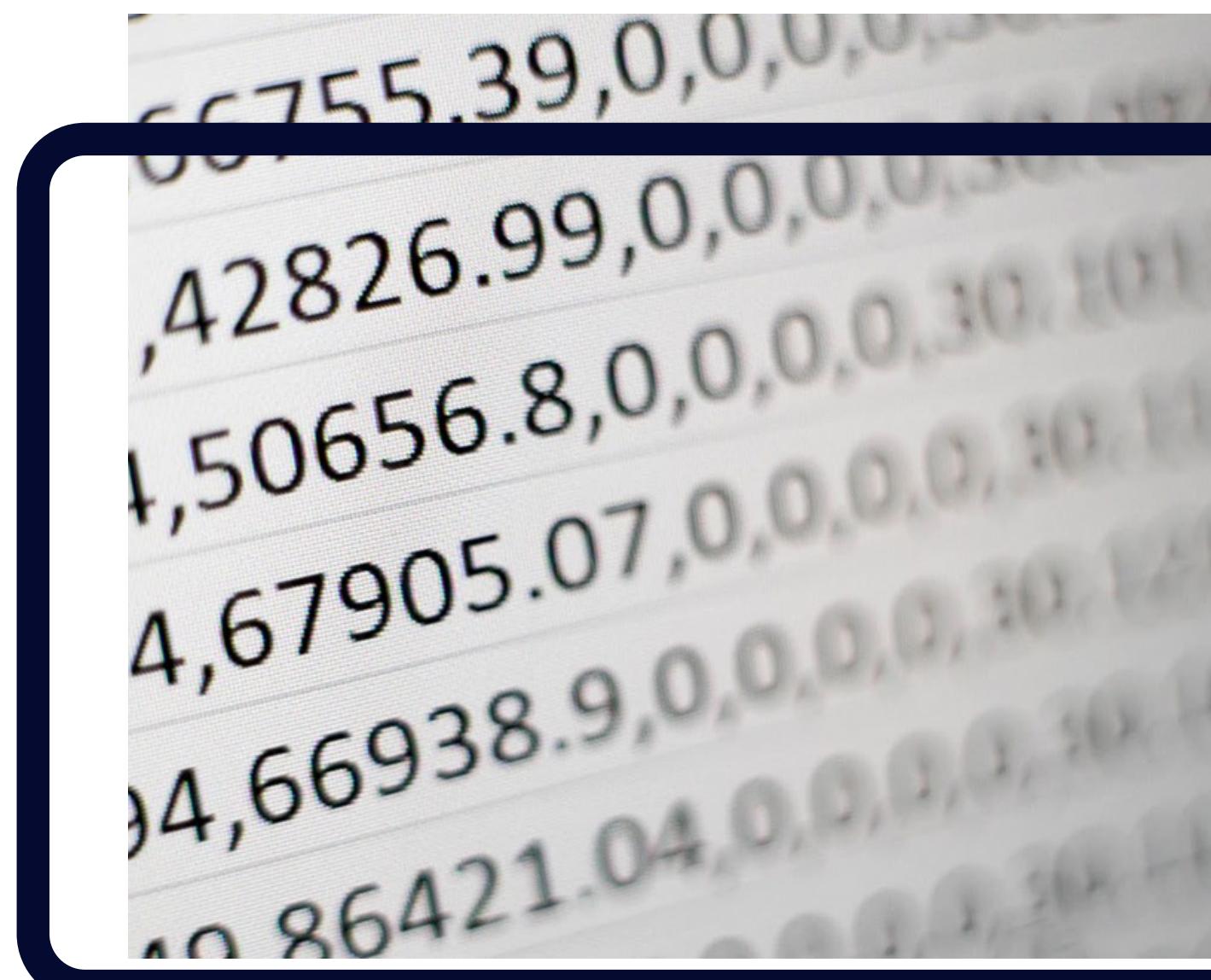
Все это может выглядеть следующим образом (обратите внимание, что я опускаю часть кода из предыдущего примера для наглядности, поэтому не забудьте добавить поле в **Meta.fields**):

```
class BorrowedFilterSet(django_filters.FilterSet):
    overdue = django_filters.BooleanFilter(method='get_overdue',field_name='returned')
    def get_overdue(self, queryset, field_name, value, ):
        if value:
            return queryset.filter(when_lte=pendulum.now().subtract(months=2))
    return queryset
```

В качестве домашнего задания попробуйте упростить этот фрагмент, добавив фильтрацию просроченных предметов аренды в **QuerySet/Model Manager** модели, как в третьей главе. Вы можете найти решение в нашем репозитории.

Конечно, существует больше типов фильтров, и самый простой способ изучить их - это прочитать документацию .

В следующей главе я собираюсь разобраться с функциональными конечными точками и вложенностью.



<https://github.com/sunscrapers/restonomicon> #репозиторий автора
<https://github.com/jumabekova06/Guide-to-the-Django-Rest-API>
#репозиторий переводчика

ГЛАВА 6: ФУНКЦИОНАЛЬНЫЕ КОНЕЧНЫЕ ТОЧКИ И ВЛОЖЕННОСТЬ API

В принципе, функциональные конечные точки (которые в **Django Rest Framework** называются действиями) выполняют задачи, которые не попадают под **CRUD** - например, отправка запроса на сброс пароля.

На протяжении всего этого руководства мы создавали приложение, позволяющее управлять предметами, взятыми во временное пользование. Сегодня мы создадим функциональную конечную точку, которая будет отправлять напоминание людям, о долгившим наши предметы.

Действия

Давайте дополним нашу модель **Friend** полем, содержащим адрес электронной почты:

Мы можем выполнить эту задачу относительно просто в методе '`get_queryset`'.

`# models.py`

```
class Friend(OwnedModel):
    email = models.EmailField(default="")
```

и обновить базу данных:

```
$ python manage.py makemigrations && python manage.py migrate
```

Теперь мы можем приступить к завершению работы над представлением кредита. Мы украсим функциональные конечные точки с помощью декоратора `@action`. Кстати, мы можем решить, будет ли действие применяться к одному элементу или ко всему списку - если да, то изменится путь, на котором появится конечная точка.

Ниже мы рассмотрим оба случая.

Сначала рассмотрим вариант с одним элементом. Мы хотим напомнить нашему другу о конкретном кредите, отправив ему электронное письмо. Мы хотим сделать это после отправки запроса **POST** в нужную конечную точку. Пример реализации выглядит следующим образом:

```
from rest_framework.decorators import action
class BorrowedViewset(viewsets.ModelViewSet):

    @action(detail=True, url_path='remind', methods=['post'])
    def remind_single(self, request, *args, **kwargs):
        obj = self.get_object()
        send_mail(subject=f"Please return my belonging: {obj.what.name}",
                  message=f"You forgot to return my belonging: '{obj.what.name}' that you
borrowed on {obj.when}. Please
return it.",
                  from_email="me@example.com", # your email here
                  recipient_list=[obj.to_who.email],
```

```
        fail_silently=False
    )
    return Response("Email sent.")
```

Определенная конечная точка появится по пути **/api/v1/borrowings/{id}/remind/**. Давайте проверим, правильно ли отправлено письмо. Если вы не хотите настраивать фактическую отправку писем, используйте следующую конфигурацию, которая пригодится во время экспериментов:

```
# settings.py
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

Теперь мы можем протестировать нашу конечную точку. В моей локальной среде кредит с **id=1** еще не был возвращен. Скорректируйте пример в соответствии с вашим конкретным примером:

Наш **API** защищен, и нам нужно отправлять авторизованные запросы. Вернитесь ко второй главе, чтобы узнать об этом подробнее.

```
$ curl -X POST http://127.0.0.1:8000/api/v1/borrowings/1/remind/
-H 'Authorization: Token fe9a080cf91acb8ed1891e6548f2ace3c66a109f'
```

Результатом должно быть "**Email** отправлено". В консоли, где запущен ваш сервер, вы должны получить сообщение, подобное следующему:

```
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: Please return my belonging: Snatch
From: me@example.com
To: friend@example.com
Date: Thu, 25 Apr 2019 08:13:23 -0000
Message-ID: <155618000311.9173.14157612795944796688@mindstorm>
You forgot to return my belonging: "Snatch" that you borrowed on 2018-01-01 12:00:00+00:00. Please return it.
```

```
[25/Apr/2019 08:13:23] "POST /api/v1/borrowings/1/remind/ HTTP/1.1" 200 13
```

Во втором случае необходимо отправить напоминание всем должникам сразу. Конечная точка должна быть доступна по этому адресу:

/api/v1/belongings/remind/ и следующий декоратор должен сделать все необходимое:

```
@action(detail=False, url_path='remind', methods=['post'])
def remind_many(self, request, *args, **kwargs):
```

Реализуйте этот метод в качестве домашнего задания. Функция **send_mail** вернет **1**, если письмо было отправлено, поэтому вы можете легко подсчитать количество отправленных писем и вернуть его в ответе.

Вложенные

Теперь перейдем к немного более продвинутой теме: вложенные конечные точки. Эта функциональность не является частью **Django Rest Framework**, и некоторые руководства советуют не использовать ее. Однако, на мой взгляд, это очень удобный способ фильтрации данных.

Несколько библиотек реализуют вложенность данных - вы можете найти список в документации **DRF**. Лично мне больше всего нравится **drf-extensions**, потому что это компактный пакет, содержащий множество полезных вещей, а не только вложенность.

В примере ниже мы составим список всех предметов, взятых на время определенным человеком. Начнем с установки библиотеки:

```
# views.py
from rest_framework_extensions.mixins import NestedViewSetMixin
# ...
class FriendViewSet(NestedViewSetMixin, viewsets.ModelViewSet):
# ...
# ...
class BorrowedViewSet(NestedViewSetMixin, viewsets.ModelViewSet):
# ...
# ...
```

Далее нам нужно расширить **Router** в файле, определяющем структуру API:

```
# api.py
from rest_framework import routers
from rest_framework_extensions.routers import NestedRouterMixin
from rentall import views as myapp_views

class NestedDefaultRouter(NestedRouterMixin, routers.DefaultRouter):
    pass
router = NestedDefaultRouter()
```

Модифицированный маршрутизатор позволяет давать имена конкретным путям и записывать в них вложенности:

```
# api.py
router = NestedDefaultRouter()
friends = router.register(r'friends', myapp_views.FriendViewSet)
friends.register(
    r'borrowings', myapp_views.BorrowedViewSet,
    base_name='friend-borrow',
    parents_query_lookups=['to_who']
)
```

Как видите, синтаксис похож на стандартный регистр. В начале указывается имя конечной точки и представление, которое ее поддерживает. Параметры **base_name** и **parents_query_lookups** являются новыми.

Первый определяет базу для имен **url**, если мы хотим использовать функцию **reverse()**. Второй содержит список полей относительно предыдущих моделей в списке. Значения, полученные из **url**, будут связаны с этими именами и использованы в качестве параметра метода **filter()** - в нашем случае это будет выглядеть следующим образом:

```
queryset = Borrowed.objects.filter(to_who={значение, полученное из url}).
```

Теперь мы можем проверить, какие предметы были одолжены одним из наших друзей:

```
$ curl http://127.0.0.1:8000/api/v1/friends/1/borrowings/
```

```
[{"id":1,"what":1,"to_who":1,"when":"2018-01-01T12:00:00Z","returned":null},  
 {"id":3,"what":1,"to_who":1,"when":"2019-04-  
 17T06:35:22.000236Z","returned":null},  
 {"id":4,"what":1,"to_who":1,"when":"2019-04-17T06:35:  
 36.546848Z","returned":null}]
```

Стоит упомянуть еще один аспект вложенности: отображение связанных моделей. Как мы получаем список одолженных предметов так, чтобы сразу отображались детали связанных моделей - например, кто одолжил предмет (**имя, email**) и что это был за одолженный предмет (**имя**).

И это будет темой следующей главы.



ГЛАВА 7: Выборочные поля и связанные объекты

Я хочу представить эту тему в двух вариантах.

Первый будет следовать порядку предыдущих глав. Второй подход разрушит этот порядок - и затем приведет к новому. Но не будем забегать вперед. ;)

Выборочные поля

Для этой функции нам понадобится библиотека **drf-dynamic-fields**:

```
$ pip install drf-flex-fields
```

Далее мы заменим текущий класс сериализатора (также можно использовать миксин).

```
# serializers.py
from rest_flex_fields import FlexFieldsModelSerializer
from rest_framework import serializers
from . import models
class FriendSerializer(FlexFieldsModelSerializer):
    owner = serializers.HiddenField(
        default=serializers.CurrentUserDefault()
    )
    class Meta:
        model = models.Friend
        fields = ('id', 'name', 'owner', 'has_overdue')
```

С этого момента мы можем указывать на конкретные поля, которые нам нужны:

<http://127.0.0.1:8000/api/v1/friends/?fields=id,name>

или перечислить поля, которые мы хотели бы исключить:

<http://127.0.0.1:8000/api/v1/friends/?omit=name>

```
GET /api/v1/friends/?fields=id,name
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "name": "_____"
    }
]
```

```
GET /api/v1/friends/?omit=name
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "has_overdue": false
    }
]
```

Добавление связанных объектов

Самый простой способ добавления связанных объектов - использование правильного сериализатора в поле **relation**.

```
# serializers.py
class BorrowedSerializer(FlexFieldsModelSerializer):
    what = BelongingSerializer()
    to_who = FriendSerializer()
    class Meta:
        model = models.Borrowed
        fields = ('id', 'what', 'to_who', 'when', 'returned')
```

Но мы должны помнить о двух вещах:

1. Чтобы не нагружать нашу базу данных, давайте заполним стандартный набор запросов **select_related**:

```
class BorrowedViewSet(NestedViewSetMixin, viewsets.ModelViewSet):
    queryset = models.Borrowed.objects.all().select_related('to_who', 'what')
```

2. Соотношения, определенные таким образом, по умолчанию предназначены только для чтения и добавления, вариант сохранения потребует дополнительной работы. Однако мы можем избежать этого, используя библиотеку, которая позволяет расширять поля, когда это необходимо:

Сначала мы укажем на сериализаторы, которые будут использоваться для расширения полей.

```
# serializers.py
class BorrowedSerializer(FlexFieldsModelSerializer):
    expandable_fields = {
        'what': (BelongingSerializer, {'source': 'what'}),
        'to_who': (FriendSerializer, {'source': 'to_who'})}
    class Meta:
        model = models.Borrowed
        fields = ('id', 'what', 'to_who', 'when', 'returned')
```

Далее нам нужно поработать с набором представлений. На этом этапе мы должны изменить базовый класс (также можно использовать миксин) и добавить список расширяемых полей.

#от переводчика:
если возникнут ошибки в коде смотрите код на **github** :
в некоторых моментах могут возникнуть ошибки.

<https://github.com/jumabekova06/Guide-to-the-Django-Rest-API/blob/master/rentall/serializers.py>

ПРИМЕЧАНИЕ: Обязательно помните о **select_related** в **queryset**. Это значительно снизит нагрузку на базу данных. Вы можете получить аналогичный результат при использовании **prefetch_related** для связей **ManyToMany** связи.

Теперь мы можем расширить наши поля.

```
GET /api/v1/borrowings/?expand=what,to_who
```

HTTP 200 OK

Allow: GET, POST, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

[

{

```
    "id": 1,
    "what": {
        "id": 1,
        "name": "Ручку"
    },
    "to_who": {
        "id": 1,
        "name": "_____н",
        "has_overdue": false
    },
    "when": "2022-02-28T16:23:20.757063Z",
    "returned": "2022-02-28T16:23:16Z"
}
```

]

CHAPTER 7: Try another approach

Наконец, я хотел показать вам еще один подход, который разрушает порядок, построенный нами до сих пор. Я собираюсь рассказать об этом, используя одно из самых хорошо разработанных расширений **DRF - DREST (Dynamic REST)**. Код для этого раздела вы можете найти в отдельной ветке нашего репозитория под названием **drest** и под тегом **part07-drest**.

Документация по **DREST** очень обширна, поэтому я остановлюсь только на наиболее важных фрагментах, относящихся к нашей теме.

Установка и конфигурация:

```
$ pip install dynamic-rest
```

```
INSTALLED_APPS = [
    .....
    "dynamic_rest",
]
```

Одна из первых вещей, которую мы должны добавить после установки, это заполнение нашего просматриваемого **API** списком всех доступных конечных точек:

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

Чтобы использовать все преимущества **DREST**, мы должны изменить используемый в настоящее время класс **ModelSerializer** на **DynamicModelSerializer**. Мы также изменим класс **ModelViewSet** на **DynamicModelViewSet**.

```
# serializers.py
from rest_framework import serializers
from dynamic_rest.serializers import DynamicModelSerializer
# ...
class FriendSerializer(DynamicModelSerializer):
# ...
class BelongingSerializer(DynamicModelSerializer):
# ...
class BorrowedSerializer(DynamicModelSerializer):
# ...
```



```
# views.py
import django_filters
from django.core.mail import send_mail
from dynamic_rest.viewsets import DynamicModelViewSet
# ...
class FriendViewSet(NestedViewSetMixin, DynamicModelViewSet):
# ...
class BelongingViewSet(DynamicModelViewSet):
# ...
class BorrowedViewSet(NestedViewSetMixin, DynamicModelViewSet):
# ..
```

К сожалению, на момент написания этой статьи библиотека несовместима с вложенными маршрутизаторами. Однако она частично реализует эту функцию автоматически, позволяя нам просматривать отношения **ManyToMany**.

Ради структуры давайте избавимся от кода, отвечающего за вложенность:

```
# api.py
from dynamic_rest.routers import DynamicRouter
from rest_framework_extensions.routers import NestedRouterMixin
from core import views as myapp_views
router = DynamicRouter()
friends = router.register(r'friends', myapp_views.FriendViewSet)
router.register(r'belongings', myapp_views.BelongingViewSet)
router.register(r'borrowings', myapp_views.BorrowedViewSet)
```

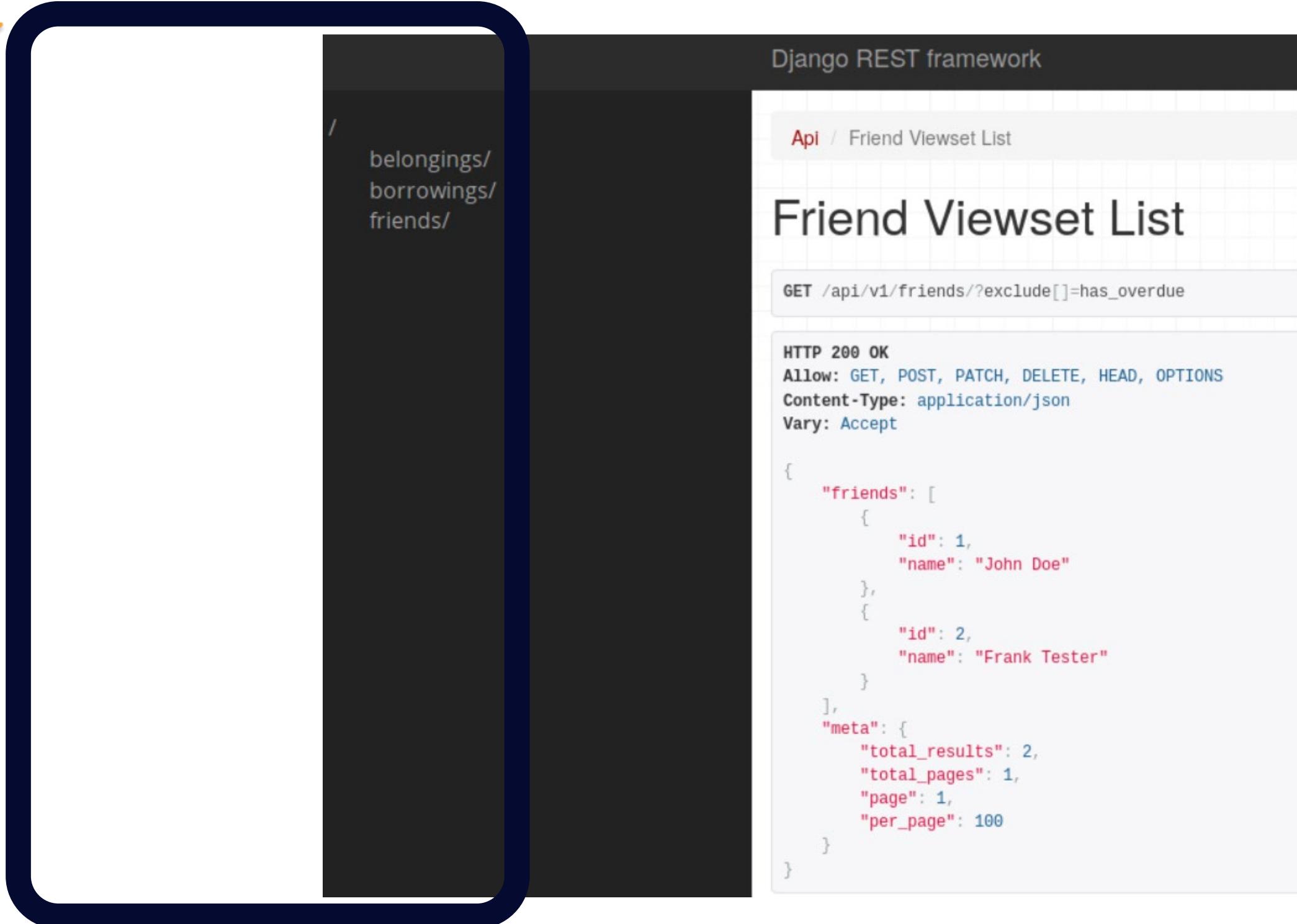
Теперь мы можем приступить к просмотру доступных функций:

Выборочные поля

Вычисление значений определенных полей в наших сериализаторах может быть очень затратным. Поэтому мы можем захотеть опустить их в определенных случаях или добавлять их только по запросу. DREST позволяет реализовать оба этих пути.

Удалить ненужные нам поля очень просто. Достаточно добавить параметр **exclude[]** в наш запрос вместе с именем поля.

[http://127.0.0.1:8000/api/v1/friends/?exclude\[\]=has_overdue](http://127.0.0.1:8000/api/v1/friends/?exclude[]=has_overdue)



Стоит обратить внимание на изменение формата возвращаемых данных по умолчанию с помощью **DynamicModelViewSet**.

Такая структура соответствует лучшим практикам **REST**.

Что касается полей, добавляемых по запросу, то мы можем отметить их с помощью параметра **deferred_fields**:

#serializers.py

```
class FriendSerializer(DynamicModelSerializer):
    owner = serializers.HiddenField(default=serializers.CurrentUserDefault())
    class Meta:
        model = models.Friend
        fields = ("id", "name", "owner", "has_overdue")
        deferred_fields = ("has_overdue",)
```

Вот как мы добавляем поле в ответ:

http://127.0.0.1:8000/api/v1/friends/?include[]>has_overdue

Еще один ключевой момент...

Добавление связанных объектов

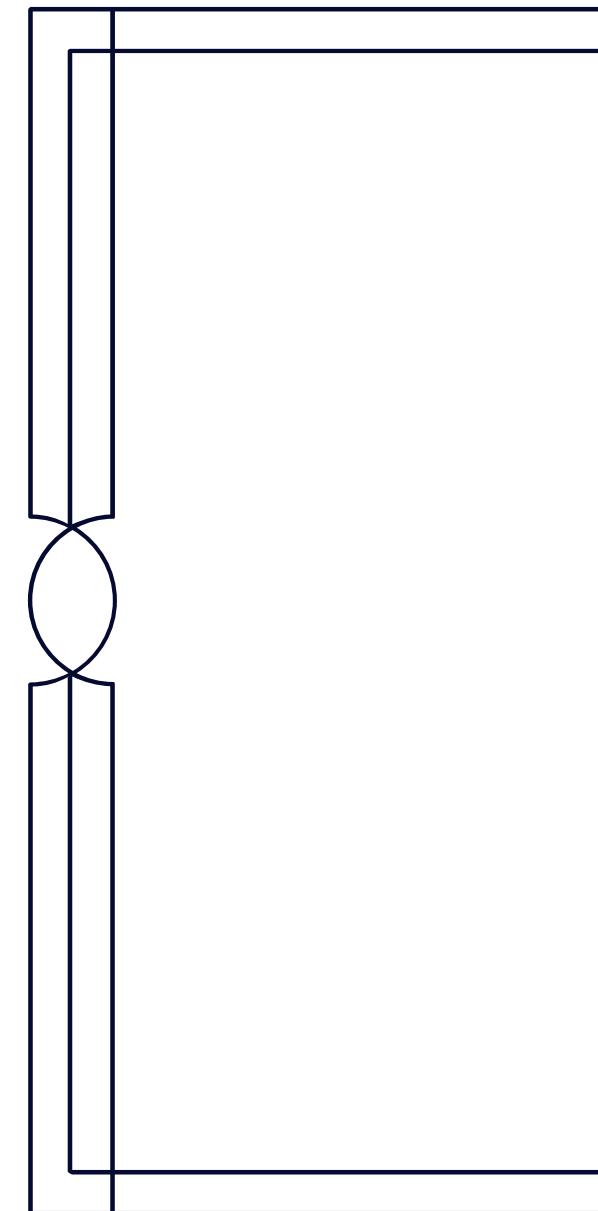
DREST поставляется с удобной функциональностью, которая дополнительно оптимизирует время выполнения запроса.

Каждая связь, которую мы отмечаем с помощью поля **DynamicRelationField**, может быть расширена, и список связанных элементов будет возвращен вместе с исходным результатом.

Например, следующий запрос включит в результаты друзей и предметы:

http://127.0.0.1:8000/api/v1/borrowings?include[]>to_who.*&include[]>what.*.

```
GET /api/v1/borrowings?include[]=to_who.*&include[]=What.*  
  
HTTP 200 OK  
Allow: GET, POST, PATCH, DELETE, HEAD, OPTIONS  
Content-Type: application/json  
Vary: Accept  
  
{  
    "belongings": [  
        {  
            "id": 1,  
            "name": "Snatch"  
        },  
        {  
            "id": 2,  
            "name": "Boondock Saints"  
        }  
    ],  
    "friends": [  
        {  
            "id": 1,  
            "name": "John Doe"  
        },  
        {  
            "id": 2,  
            "name": "Frank Tester"  
        }  
    ],  
    "borroweds": [  
        {  
            "id": 1,  
            "what": 1,  
            "to_who": 1,  
            "when": "2018-01-01T12:00:00Z",  
            "returned": null  
        },  
        {  
            "id": 2,  
            "what": 2,  
            "to_who": 2,  
            "when": "2019-04-16T18:46:16.646649Z",  
            "returned": "2019-04-16T18:46:13Z"  
        }  
    ]  
},
```



Да, я понимаю, что имя заимствованных объектов по умолчанию не так уж и хорошо...

Если наше приложение имеет более сложную структуру, мы можем использовать этот метод для включения объектов на любом уровне вложенности. Однако нам нужно помнить о нагрузке, которую мы возлагаем на базу данных, поскольку она становится больше с каждым уровнем.

Эта глава завершает мою серию статей о **Django REST Framework**, в которой я затронул практически все аспекты работы с **REST API**.

Просто чтобы напомнить вам, вот ссылка на репозиторий, где вы найдете весь код, который я обсуждал в этой электронной книге.

Если вышеизложенного вам недостаточно, у вас есть два варианта. Вы можете либо проложить свой собственный путь и поделиться опытом с другими, либо обратиться к **GraphQL** (но это тема для другой электронной книги).

Но пока... Пока и спасибо за всю рыбу!

#ссылка на репозиторий автора:
<https://github.com/sunscrapers/restonomicon/>

#ссылка на репозиторий переводчика:
<https://github.com/jumabekova06/Guide-to-the-Django-Rest-API>



Хотите стать частью нашей команды?

<http://sunscrapers.com/careers/>



Unrivaled Python Engineers

Sunscrapers Sp. z o.o.
Pokorna 2/947 (entrance 9)
00-199 Warsaw

New Business
hello@sunscrapers.com
Careers
[careers@sunscrapers.com](mailto:ccareers@sunscrapers.com)

