DJANGO. DJANGO REST FRAMEWORK

Оглавление

[1 Зачем нужны фреймворки 2](#_Toc133442930)

[2 Содержимое директории проекта 3](#_Toc133442931)

[3 Содержимое директории приложения 3](#_Toc133442932)

[4 Django ORM и модели 4](#_Toc133442933)

[5 Суперпользователь. Админка в Django 6](#_Toc133442934)

[6 Обработка запросов в Django 8](#_Toc133442935)

[7 Преобразование форматов. Сериализаторы 9](#_Toc133442936)

[8 View-функции API 12](#_Toc133442937)

[9 View-классы API 17](#_Toc133442938)

[10 Вьюсеты и роутеры 20](#_Toc133442939)

[11 Сериализаторы для связанных моделей 24](#_Toc133442940)

[12 Сериализаторы: дополнительные настройки 32](#_Toc133442941)

[13 Регулярные выражения 35](#_Toc133442942)

[14 Вьюсеты. Расширенные возможности 37](#_Toc133442943)

[14 Аутентификация по токену. JWT + Djoser 41](#_Toc133442944)

[14.1 Аутентификация через Authtoken 42](#_Toc133442945)

[14.2 Аутентификация по JWT-токену 44](#_Toc133442946)

[15 Kittygram 2. Валидация 48](#_Toc133442947)

[16 Проверка прав: Permissions 55](#_Toc133442948)

[17 Пагинация в API 61](#_Toc133442949)

# 1 Зачем нужны фреймворки

В совершенно разных проектах разработчики вынуждены решать стандартные задачи. Раз за разом программисты пишут системы хранения и модификации данных, инструменты для управления аккаунтами, программы для отображения информации на экране пользователя. Эти системы есть почти в каждом проекте, также как у любого дома есть стены, крыша и окна.

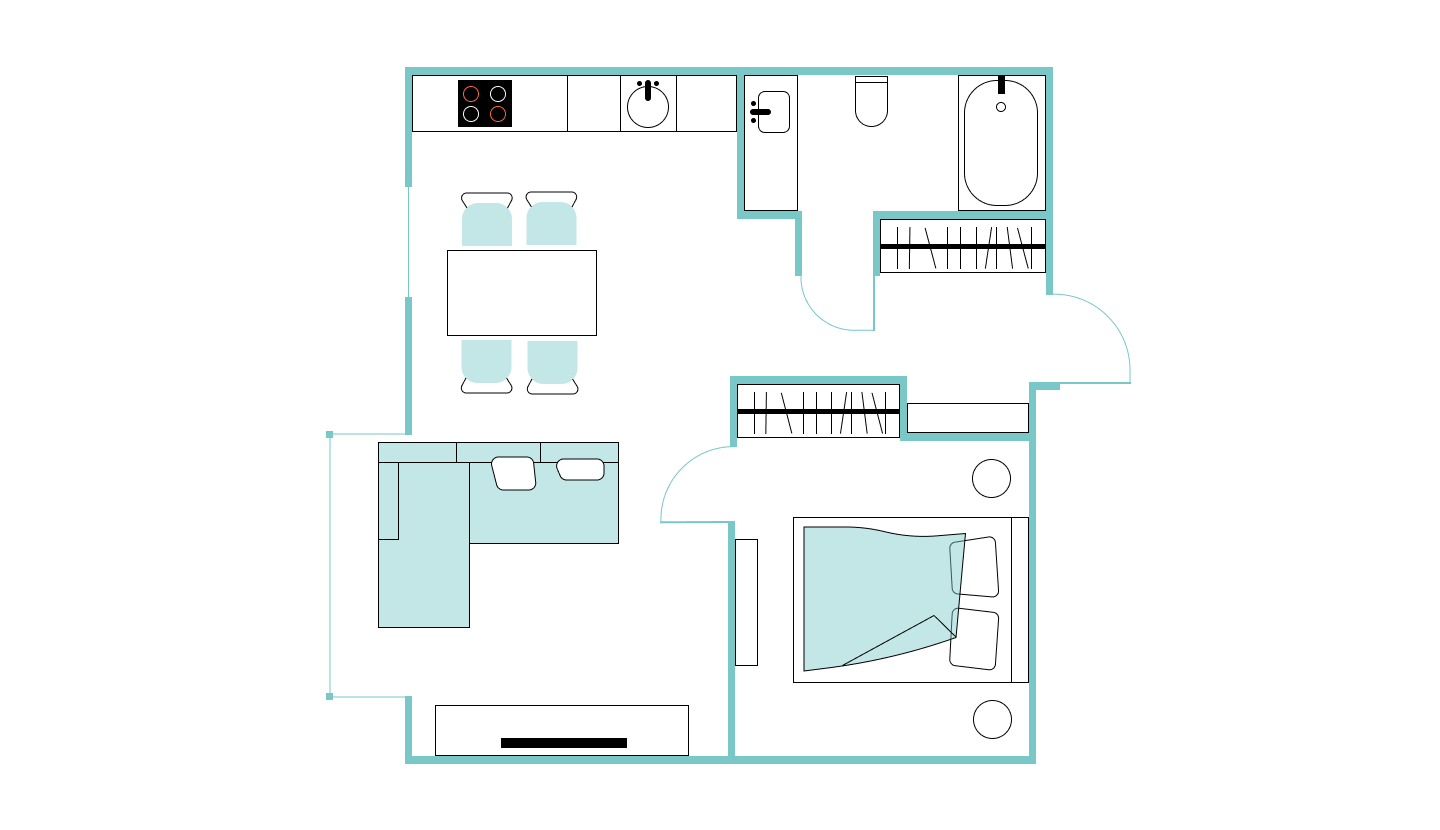
Предположим, нужно обзавестись жильём.

Можно сделать всё самостоятельно. Изготовить кирпичи и сложить стены. Напилить досок и сделать окна, двери и табуретки. Не забыть бы про плитку в ванной, трубы и тысячу других вещей.

Долго. Дорого. Неэффективно. Сложно в обслуживании (никто не знает, как у вас всё устроено).

А можно купить готовую квартиру: у неё есть внешние стены, подведена вода, отопление, канализация и свет. Останется установить внутренние стены, наклеить обои и поставить мебель.

Такая квартира — это фреймворк, основа проекта. Перенести несущие стены или сделать из неё самолет или корабль не получится, но всё, что внутри, можно менять в довольно широких рамках. Мраморный пол, махровый халат в ванной и мягкий кот? На здоровье. Чёрные стены, кованые шторы и свирепый доберман? Пожалуйста.

Можно сделать всё, что угодно. Фреймворк позволит быстро обустроиться и жить с комфортом.

В разработке всё устроено аналогично. Можно всё сделать самому, а можно взять фреймворк, который создаст каркас файловой структуры, подключит полезные библиотеки, предоставит возможность применить модули других разработчиков. В результате задача будет решена качественно и без лишних усилий.

На Python написана масса фреймворков для распознавания изображений или голоса, для работы с большими данными, для создания игр и мобильных приложений, для получения данных с сайтов и их обработки — список можно продолжать долго.

В этом курсе вы будете работать с фреймворком Django — одним из наиболее популярных фреймворков для веб-разработки на Python.

**За что любят Django**

**Django** работает, например, в *Instagram*, *Mozilla*, *The Washington Times*, Pinterest, *National Geographic* и тд.

* **Экосистема и расширяемость**

В Django входит большое количество сторонних приложений, типовых блоков, как в конструкторе Лего. В [официальном каталоге](https://djangopackages.org/) есть сотни плагинов и библиотек, которые помогут быстро и качественно разработать проект: в большинстве случаев нужно лишь выбрать подходящую библиотеку и, если потребуется, немного доработать код.

* **Сообщество**

Django появился в 2005 году. С тех пор тысячи специалистов решили сотни тысяч задач и поделились своим опытом в интернете. Если возник вопрос — стоит как следует поискать в сети, и ответ найдется. Поиск ответов на вопросы — одна из важных составляющих в профессии разработчика, и при работе в Django такой поиск всегда даст результат.

* **Настраиваемая админка**

Система управления информацией (админ-зона проекта, «админка») в Django создаётся автоматически. При необходимости её можно настроить в очень широких пределах.

* **Работа с базами данных**

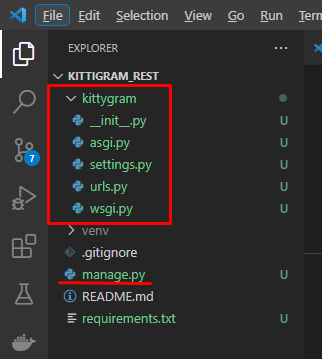
С базами данных Django общается через инструмент Django ORM (объектно-реляционное отображение). Это «переводчик» с языка Python на язык SQL, понятный большинству баз данных. Благодаря Django ORM все запросы к базе данных можно писать прямо на Python.

# 2 Содержимое директории проекта

После выполнения команды создания базовой структуры проекта:

*$ django-admin startproject kittygram .*

вы получите следующую структура проекта:



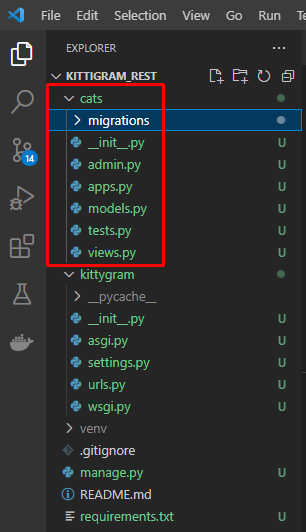
* В директории ***kittygram\_rest/kittygram*** лежат файлы с кодом проекта.
* ***kittygram\_rest/manage.py*** — файл управления Django-проектом из командной строки. Вы часто будете к нему обращаться.
* В файле ***kittygram\_rest/kittygram/urls.py*** настраиваются URL проекта.
* ***kittygram\_rest/kittygram/wsgi.py*** — это файл конфигурации WSGI-сервера, он пригодится при размещении проекта на веб-сервере.
* В файле ***kittygram\_rest/kittygram/settings.py*** хранятся все настройки проекта. При развёртывании проекта автоматически устанавливаются стандартные настройки, а в течение развития проекта разработчик изменяет или дополняет их.

# 3 Содержимое директории приложения

После выполнения команды создания нового приложения:

*$ python manage.py startapp cats*

вы получите следующую структуру проекта:



* **admin.py** — здесь можно настроить отображение админ-зоны приложения.
* **apps.py** — настройки конфигурации приложения.
* **models.py** — здесь разработчик описывает устройство базы данных приложения.
* **migrations/** — тут хранится история изменений в базе данных.
* **tests.py** — файл для тестов приложения.
* **views.py** — тут хранятся обработчики запросов (функции или классы, получающие запрос и генерирующие ответ).

# 4 Django ORM и модели

Классы описывают новые типы объектов и позволяют создавать экземпляры таких объектов. Записи в базах данных тоже описывают объекты — наборы свойств, которыми можно управлять.

Есть способ связать данные объектов с записями в БД, упростить и автоматизировать стандартные операции и при этом обойтись без запросов на SQL.

Всё это делает **Django ORM** — **Object-Relational Mapping**, «*объектно-реляционное отображение*». **Object** — объекты, которые созданы на основе классов, **relational** — реляционные базы данных, а **mapping** — связь между системой объектов и базами данных.

Django ORM — это инструмент для работы с данными реляционной БД посредством классов, которые создаёт сам программист. Реализаций ORM существует много, работать мы будем с ORM, встроенной в Django.

**Модели в ORM**

Классы, с которыми работает ORM, называются моделями. В Django ORM есть предустановленный класс **Model**, от которого разработчик может наследовать собственные модели. У этого класса есть множество предустановленных свойств и методов, обеспечивающих работу с БД.

Например, планируется таблица «Мороженное» со следующими полями:

* Название мороженого.
* Описание мороженого.
* Должно ли оно отображаться на главной странице

Вот как будет выглядеть описание данной таблице через класс Model в Django:

# Объявляем класс IceCream, наследник класса Model из пакета models

class IceCream(models.Model):

    # Описываем поля модели и их типы

    # Тип: CharField (строка с ограничением длины)

    name = models.CharField(max\_length=200)

    # Тип: TextField (текстовое поле)

    # используется для больших текстовых блоков

    description = models.TextField()

    # Тип: булева переменная, «да/нет»;

    # значение по умолчанию: True

    on\_main = models.BooleanField(default=True)

Магия ORM состоит в том, что после создания модели Django автоматически проведёт массу операций:

* Создаст необходимые таблицы в базе данных.
* Добавит первичный ключ (primary key), по которому можно будет обратиться к нужной записи.
* Добавит интерфейс администратора.
* Создаст формы для добавления и редактирования записей в таблице.
* Настроит проверку данных, введённых в веб-формы.
* Предоставит возможность изменения таблиц в БД.
* Создаст SQL-запросы для создания таблицы, поиска, изменения, удаления данных, настроит связи между данными, обеспечив их целостность.
* Предоставит специальный синтаксис формирования запросов.
* Добавит необходимые индексы в базу данных для ускорения работы сайта.

Или вот пример описания таблицы «Post»:

from django.db import models

from django.contrib.auth import get\_user\_model

User = get\_user\_model()

class Post(models.Model):

    text = models.TextField()

    pub\_date = models.DateTimeField(auto\_now\_add=True)

    author = models.ForeignKey(User, on\_delete=models.CASCADE)

В коде модели Post описаны поля:

* **TextField** — поле для хранения произвольного текста.
* **DateTimeField** — поле для хранения даты и времени. Существуют похожие типы для хранения даты (DateField), промежутка времени (DurationField), просто времени (TimeField).
* **ForeignKey** — поле, в котором указывается ссылка на другую модель, или, в терминологии баз данных, ссылка на другую таблицу, на её primary key (pk). В нашем случае это ссылка на модель User. Это свойство обеспечивает связь (relation) между таблицами баз данных.

Параметр **on\_delete=models.CASCADE** обеспечивает связность данных: если из таблицы **User** будет удалён пользователь, то будут удалены все связанные с ним посты.

Другие популярные типы полей:

* **BooleanField** — поле для хранения данных типа bool.
* **EmailField** — поле для хранения строки, но с обязательной проверкой синтаксиса email.
* **FileField** — поле для хранения файлов. Есть сходный, но более специализированный тип ImageField, предназначенный для хранения файлов картинок.

В Django ORM есть и другие типы полей. Документация даёт полное описание базовых полей, но есть расширения, добавляющие новые типы полей или переопределяющие базовые типы.

# 5 Суперпользователь. Админка в Django

При разворачивании проекта устанавливаются необходимые приложения, в частности **django.contrib.admin** и **django.contrib.auth**. При миграции эти приложения добавили свои таблицы в базу данных:

* **admin** — создаёт интерфейс администратора сайта,
* **auth** — управляет пользователями.

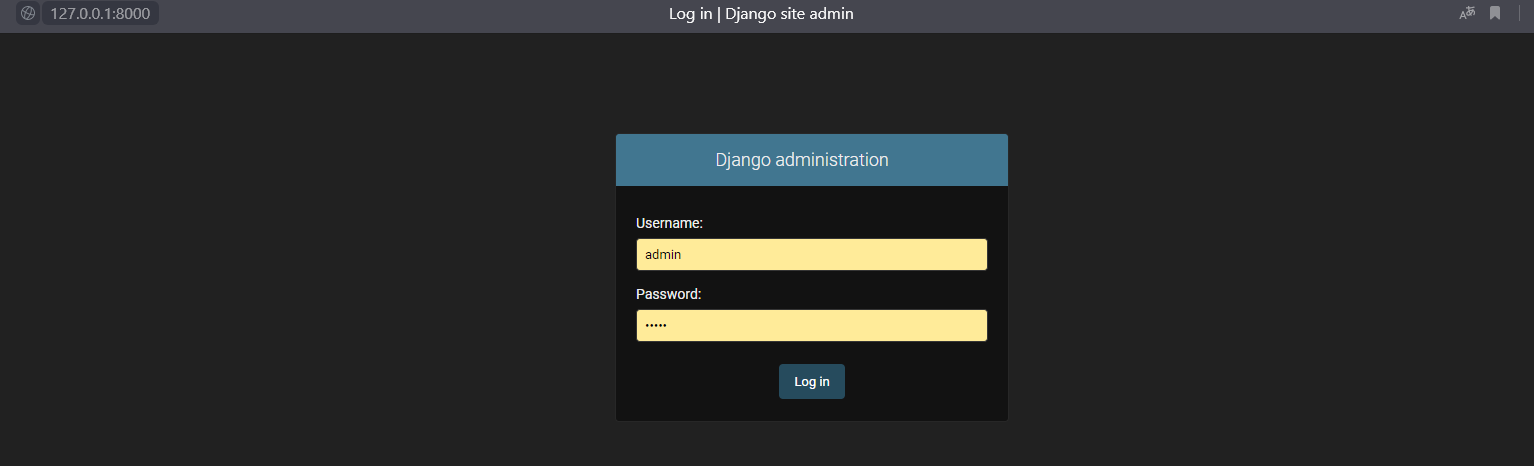
Именно эти приложения позволят нам создать учётную запись администратора сайта и авторизоваться на сайте. При создании администратора мы дадим ему максимум прав; такие аккаунты в Django называются «суперпользователями» (superuser).

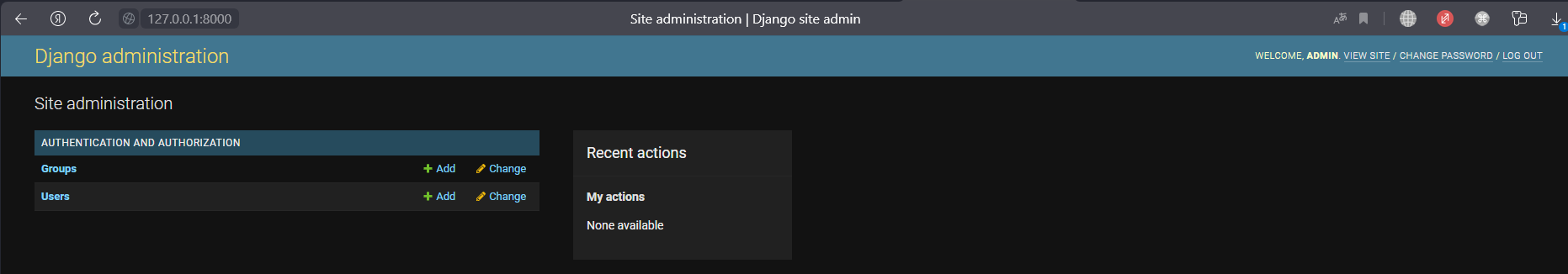
Под этой учётной записью вы будете управлять сайтом через интерфейс администрирования (админ-зону).

Для создания суперпользователя выполните команду:

*$ python manage.py createsuperuser*

Откройте в браузере адрес *http://127.0.0.1:8000/admin/.* Вы увидите страницу авторизации:





**Регистрация модели в админке**

Администратор сайта должен иметь возможность управлять публикациями (например, удалять сообщения со спамом). Можно делать это через прямые запросы к базе, но не каждый админ на это способен.

В Django предусмотрен графический интерфейс для администрирования любой модели: надо только подключить эту модель к админ-зоне.

Модели не добавляются в интерфейс админки автоматически, ведь не все они нужны администратору. По умолчанию в проекте уже есть множество моделей; вы видели, что в результате миграции в базу данных добавилось много таблиц. Но в интерфейсе админки видны только две модели: Groups и Users. Остальные модели — служебные, они не требуют внимания администратора и потому исключены из интерфейса.

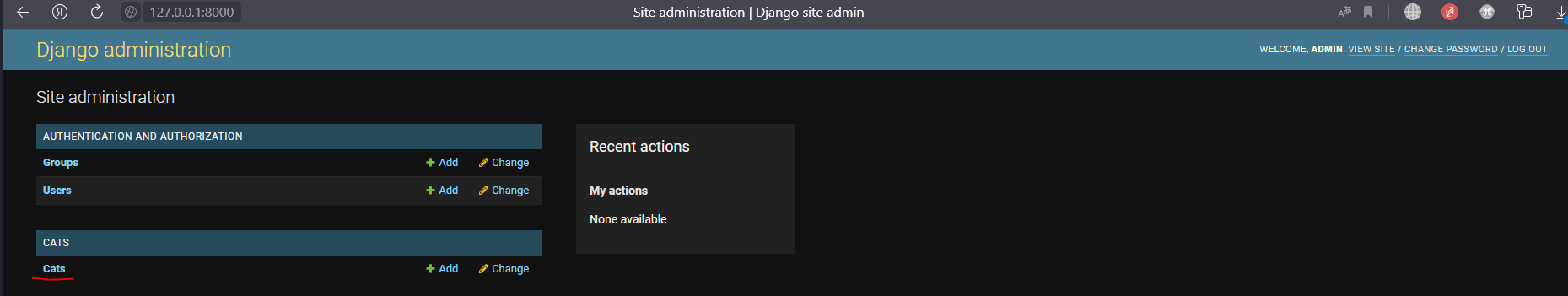
Чтобы добавить модель **Cat** в интерфейс администратора, её надо зарегистрировать в файле **cats/admin.py**.

from django.contrib import admin

from .models import Cat

admin.site.register(Cat)

Сохраните файл, перезагрузите страницу админки — и вы увидите новый раздел:



**Конфигурация модели в admin.py**

Для настройки отображения модели в интерфейсе админки применяют класс **ModelAdmin**. Он связывается с моделью и конфигурирует отображение данных этой модели. В этом классе можно настроить параметры отображения.

Добавим следующий класс и код в файл **cats/admin.py**:

from django.contrib import admin

from .models import Cat

class CatAdmin(admin.ModelAdmin):

    # Перечисляем поля, которые должны отображаться в админке

    list\_display = ('pk', 'name', 'color', 'birth\_year')

    # Добавляем интерфейс для поиска по имени

    search\_fields = ('name',)

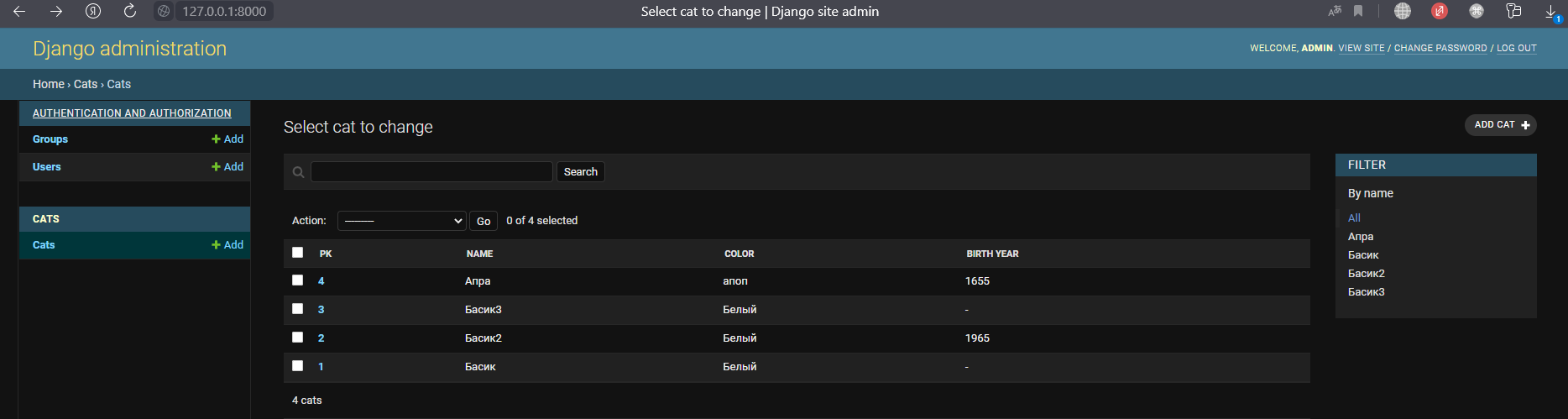
    # Добавляем возможность фильтрации по имени

    list\_filter = ('name',)

# При регистрации модели Cat источником конфигурации для неё назначаем

# класс CatAdmin

admin.site.register(Cat, CatAdmin)



Свойства, которые мы настроили:

* **list\_display** — перечень свойств модели, которые мы хотим показать в интерфейсе.
* **search\_fields** — перечень полей, по которым будет искать поисковая система. Форма поиска отображается над списком элементов.
* **list\_filter** — поля, по которым можно фильтровать записи. Фильтры отображаются справа от списка элементов.

# 6 Обработка запросов в Django

При создании проекта разработчик сам придумывает адреса, которые будут доступны на сайте. Эти адреса хранятся в коде; при запросе к проекту Django перебирает список адресов проекта, сравнивая их с запрошенным.

Если запрошенный URL есть в списке, то вызывается связанная с этим адресом функция, которая обрабатывает запрос и делает всё необходимое: генерирует и отправляет пользователю запрошенную страницу, записывает данные в БД или делает что-то ещё хорошее.

Если же запрошенный URL не обнаружился в списке заготовленных адресов, пользователю отправляется страница с сообщением об ошибке: **404**, *Page Not Found*, «вы ошиблись адресом!».

Списки адресов в Django хранят в переменной *urlpatterns* в файлах **urls.py**. Основной файл **urls.py** лежит в главной папке проекта. Адреса в списке указывают в виде относительных URL.

После того как Django найдёт совпадение запрошенного URL с шаблоном адреса из списка, должен быть вызван **обработчик** — функция или класс, в которых после обработки запроса будет подготовлен ответ.

Для связи URL и обработчика применяется функция **path()**. Она принимает обязательные параметры *path('route', view)*:

* **route** — шаблон обрабатываемого адреса, образец, с которым сравнивается полученный запрос;
* **view** — функция-обработчик: если запрошенный URL совпадает с **route**, вызов будет перенаправлен в указанную view-функцию (view-функции в Django хранят в файле **views.py**).

При получении запроса Django проходит по списку **urlpatterns** сверху вниз, пока не найдёт совпадение запрошенного адреса с шаблоном адреса. При обнаружении совпадения будет вызвана соответствующая **view-функция**.

**Разделение адресов по приложениям. Функция include()**

Обычно Django-проект состоит из нескольких приложений, и каждое приложение обрабатывает свою часть запросов.

Хранить все шаблоны адресов всех приложений в одном файле будет неудобно: в какой-то момент список шаблонов разрастётся и станет нечитаемым. Хорошей стратегией будет разделить список **urlpatterns** на части, в соответствии с приложениями, и хранить эти части в директориях приложений.

Django даёт такую возможность: в каждом приложении можно создать собственный файл **urls.py**, в котором будут перечислены адреса, обрабатываемые именно этим приложением, а в корневом **urls.py** будут ссылки на эти файлы. Для создания таких ссылок используется функция **include()** (англ. «включить», «встроить»).

**Конвертеры пути**

В адресе **cats/<pk>/** в аргументе **pk** ожидается число, но пользователь может по ошибке или из любопытства вписать туда любую строчку: например, отправить запрос **cats/barsik/** вместо **cats/15/**. Это нарушит нам весь фэншуй.

Для профилактики такого несознательного поведения используют конвертеры пути (англ. *path converters*): перед именем переменной указывается тип ожидаемых данных: **cats/<int:pk>/.**

Если в аргументе будут данные, не соответствующие конвертеру, Django будет считать, что шаблон адреса не совпадает с запросом, и продолжит поиск совпадений по urlpatterns.

Теперь ни один пользователь не сможет накормить обработчик неудобоваримой переменной.

Вот список наиболее востребованных конвертеров:

* **str** — ожидает непустую строку, состоящую из любых символов, исключая разделитель пути '/'. Если в параметрах пути конвертер не указан явно, то именно конвертер str будет применён по умолчанию: шаблон адреса user/<username>/ идентичен шаблону user/<str:username>/.
* Конвертер **int** ожидает в качестве значения ноль или любое целое положительное число. Синтаксис: cats/<int:pk>/.
* Конвертер **slug** ожидает строку из букв и цифр, входящих в стандарт ASCII, а также символов - и \_.

Обычно слаг используют для создания человекочитаемых URL. При сравнении двух URL, в одном из которых аргумент — число, а в другом — slug, станет видна разница: по адресу group/cats/ безусловно ясно, что страница про котиков. А из адреса group/1/ не понятно вообще ничего, сплошь туманная неопределённость.

# 7 Преобразование форматов. Сериализаторы

**API** — это интерфейс для программ, именно программы обмениваются информацией друг с другом. При этом программы, которые планируют обмениваться данными, могут быть написаны на разных языках программирования, но именно благодаря API они без проблем смогут общаться между собой, обмениваясь данными в удобном и едином для всех формате (например JSON, XML и т.д.).

Таким образом, чтобы описанное взаимодействие получилось, данные постоянно должны преобразовываться из формата в формат: из типов данных конкретного языка программирования в выбранный формат обмена данными и обратно. Такие преобразования получили названия: **сериализация** и **десериализация**.

**Сериализация**

Первая задача: пользователь хочет прочесть какой-то конкретный пост из своей ленты друзей. Для этого должны быть выполнены такие операции:

* Мобильное приложение отправит *GET-запрос* к **API**; в запросе будет передан **id** поста.
* API сделает запрос к базе данных проекта в надежде получить оттуда экземпляр класса **Post** с заданным **id**.
* Если пост с таким **id** есть в БД, то программа получит объект класса **Post** (этот объект содержит несколько полей разных типов данных).
* В качестве ответа на вопрос нужно вернуть данные в формате JSON. Преобразование Python-объекта в JSON происходит в два шага:
  + сложный объект (например, экземпляр модели) преобразуется в словарь, содержащий простые типы данных Python;
  + получившийся Python-словарь конвертируется («рендерится») в JSON.
* JSON отправляется в HTTP-ответе в мобильное приложение.

Например, из БД получен объект класса **Post**, который можно было бы описать так:

post = Post(

    id=87,

    author='Робинзон Крузо',

    text='23 ноября. Закончил работу над лопатой и корытом.',

    pub\_date='1659-11-23T18:02:33.123543Z'

)

После **сериализации** этого объекта будет создан такой JSON:

{

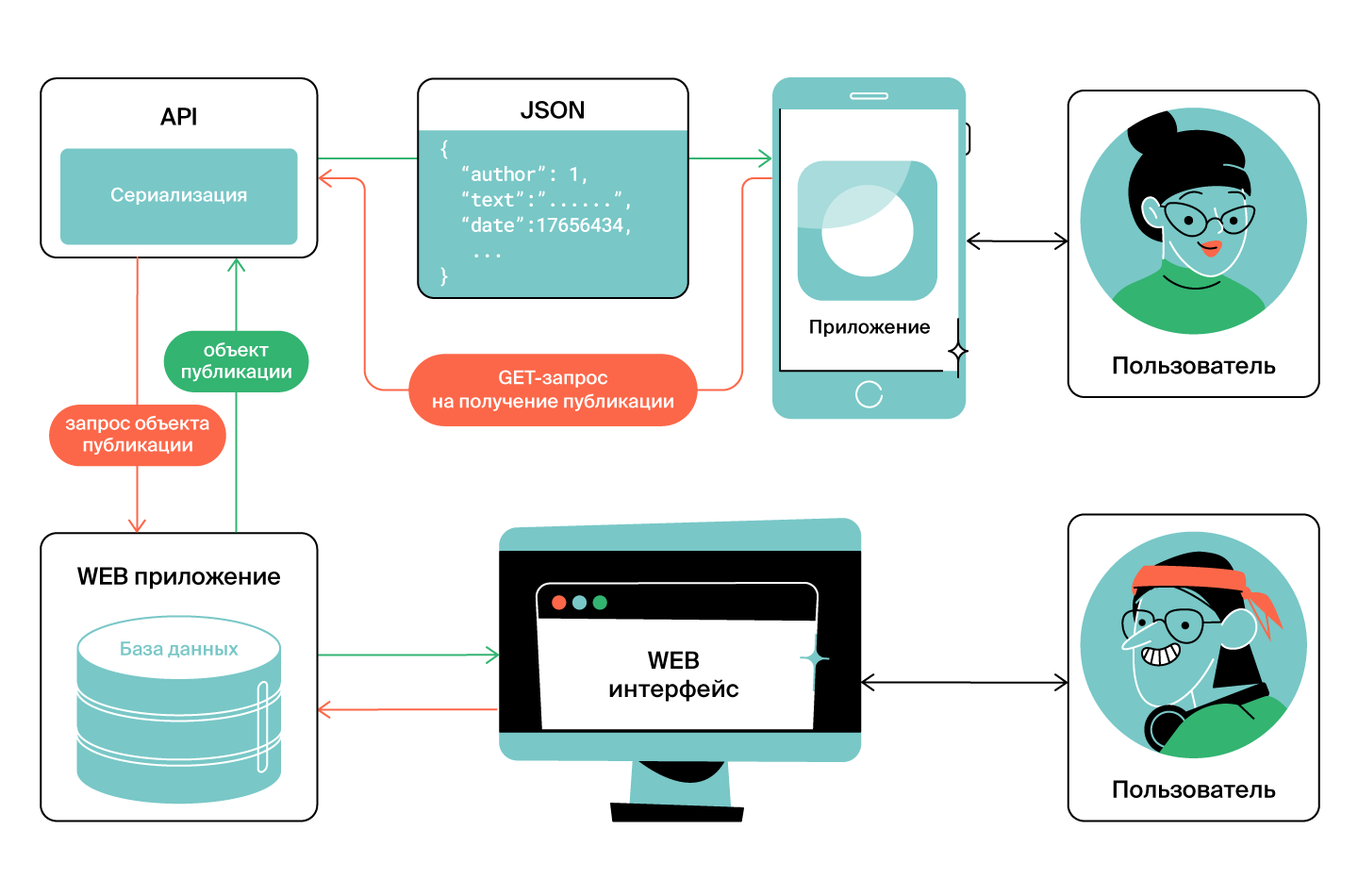
    "id": 87,

    "author": "Робинзон Крузо",

    "text": "23 ноября. Закончил работу над лопатой и корытом.",

    "pub\_date": "1659-11-23T18:02:33.123543Z"

}



**Валидация и десериализация**

Вторая задача: пользователь решил через мобильное приложение добавить в блог новую запись. Пользователь набрал текст и нажал кнопку «Отправить».

Из мобильного приложения будет отправлен *POST-запрос* к **API**, и при получении запроса API выполнит действия, подобные описанным, но в обратном направлении: преобразует JSON из запроса в Python-объект. В процесс добавится дополнительный шаг — проверка данных на корректность, **валидация**.

Будут выполнены такие операции:

* Мобильное приложение сформирует и отправит POST-запрос к API проекта; в теле запроса будет передан JSON, содержащий всю необходимую информацию для создания нового поста.
* Полученный JSON должен быть **десериализован**. **Десериализация** происходит в три этапа:
  + Преобразование JSON в простые типы данных Python.
  + **Валидация** — проверка соответствия полученных данных ожиданиям.
  + Конвертация валидированных данных в сложные объекты Python (в queryset или объект модели).
* Если полученные данные соответствуют модели, то API отправит запрос к БД, чтобы добавить в неё новый объект, а пользователю отправит ответ с подтверждением

Если же полученные данные не пройдут валидацию, то API не будет добавлять их в базу данных, а просто отправит ответ с сообщением об ошибке.

Пример JSON, который API получит в POST-запросе:

{

    "author": "Робинзон Крузо",

    "text": "24 декабря. Всю ночь и весь день шёл проливной дождь.",

    "pub\_date": "1659-12-24T21:14:56.123543Z"

}

JSON будет **десериализован** в объект Python, который можно описать так:

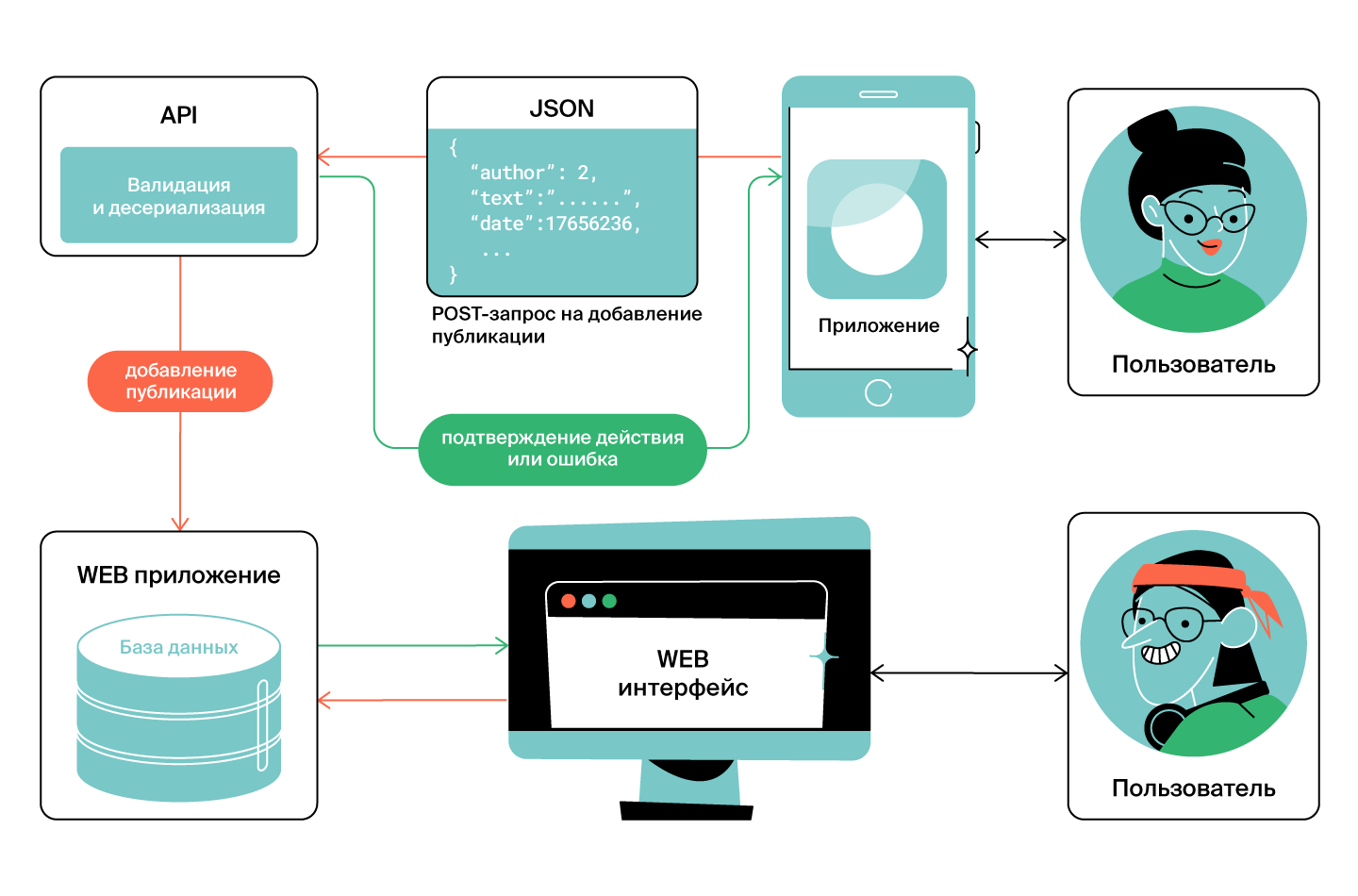
post = Post(

    author='Робинзон Крузо',

    text='24 декабря. Всю ночь и весь день шёл проливной дождь.',

    pub\_date='1659-12-24T21:14:56.123543Z'

)



**Классы-сериализаторы в DRF**

В Django REST Framework есть классы, которые способны принимать участие во всех трёх операциях: сериализации, валидации и десериализации. Эти классы называются **сериализаторы** (**serializers**).

Сериализаторы преобразуют сложные данные, такие как queryset или экземпляр модели, в простые типы данных Python, которые затем можно конвертировать («отрендерить») в JSON, XML или другие форматы обмена.

Сериализаторы выполняют и обратное преобразование: конвертируют данные, полученные из JSON, в сложные объекты; при этом данные проходят валидацию.

Чаще всего работа с данными в Django осуществляется через модели, и в таких случаях сериализатор наследуется от класса **ModelSerializer**.

При создании такого сериализатора во внутреннем классе **Meta** нужно указать модель, с которой должен работать сериализатор, и список тех полей модели, которые нужно сериализовать или десериализовать.

Для наследников ModelSerializer нет необходимости описывать типы полей и их параметры: сериализатор сам их определит, взяв за основу поля указанной модели.

В качестве примера опишем модель Comment и сериализатор для неё:

class Comment(models.Model):

    post = models.ForeignKey(Post, on\_delete=models.CASCADE)

    author = models.ForeignKey(User, on\_delete=models.CASCADE)

    text = models.TextField()

    created = models.DateTimeField('created', auto\_now\_add=True)

class CommentSerializer(serializers.ModelSerializer):

    class Meta:

        model = Comment

        # Указываем поля модели, с которыми будет работать сериализатор;

        # поля модели, не указанные в перечне, сериализатор будет игнорировать.

        # Для перечисления полей можно использовать список или кортеж.

        fields = ('id', 'post', 'author', 'text', 'created')

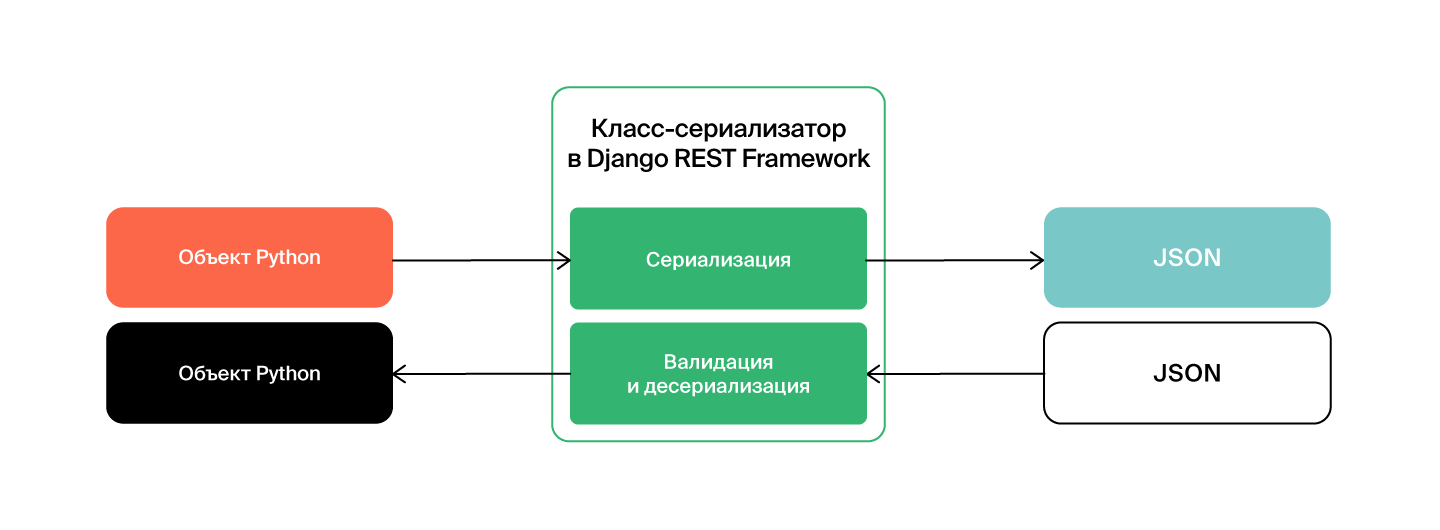
Чтобы сериализатор работал со всеми полями модели без исключения, можно указать **fields = '\_\_all\_\_'**.

Вместо fields можно применить настройку **exclude** (англ. «исключить»): в этом случае сериализатор будет работать со всеми полями модели, за исключением перечисленных.

Лучшим решением будет явно перечислять поля. Если в ходе разработки в модель будет добавлено новое поле, не предназначенное для публикации, а в сериализаторе указано **fields = '\_\_all\_\_'** — велик шанс, что содержимое нового поля попадёт в ответ API и будет обнародовано.

«Явное лучше неявного».

**Действия разные, но сериализатор один**

Один и тот же класс-сериализатор в DRF можно применять как для сериализации, так и для десериализации данных. Логика его работы будет выбрана автоматически, в зависимости от того, какой именно объект передан в сериализатор.

Если в конструктор сериализатора передать экземпляр класса, то будет запущен процесс сериализации. А если в качестве параметра передать данные из JSON-объекта, то будет запущен процесс десериализации.

Код классов-сериализаторов принято выносить в отдельный файл **serializers.py**.

# 8 View-функции API

В Django REST Framework запрос к API передается нужному представлению в соответствии с адресами, перечисленными в файле *urls.py*.

Для обработки запросов к API могут использоваться представления как в виде **функций**, так и в виде **классов**. Функции, как правило, позволяют лучше понять как работает код; вот с них и начнём.

Для «настройки» view-функции на работу с API в Django REST framework есть декоратор **@api\_view**. В качестве аргумента декоратору передают список типов HTTP-запросов, которые должна обрабатывать эта функция:

@api\_view(['GET', 'POST'])

def cat\_list(request):  # Применили декоратор и указали разрешённые методы

Ещё одно отличие view-функции API в Django REST framework состоит в том, что они возвращают специальный объект класса **Response**; в этот объект в качестве аргумента передаётся Python-словарь, данные из которого и должны быть отправлены в ответ на запрос в JSON формате.

cats = Cat.objects.all()

    serializer = CatSerializer(cats, many=True)

    return Response(serializer.data)

**View-функция API: обработка POST-запроса с одним объектом**

Начнём с создания записи в базе данных об одном котике через API.

Модель **Cat** содержит несколько полей:

* имя котика: **name**,
* его цвет: **color**,
* его год рождения: **birth\_year**.

Чтобы добавить в БД запись о новом котике, нужно отправить POST-запрос на эндпоинт *cats/*. В теле POST-запроса должен быть передан объект в формате JSON; содержание этого объекта должно соответствовать модели Cat.

JSON, переданный в POST-запросе, может быть таким:

# Тело POST-запроса

{

    "name": "Стёпа",

    "color": "белый",

    "birth\_year": 1971

}

Прежде чем сохранить полученные данные в БД, их нужно десериализовать: удостовериться, что они соответствуют ожиданиям модели Cat и конвертировать из простых типов данных в объект.

Для этого потребуется сериализатор; он уже описан в проекте, это класс **CatSerializer**.

from rest\_framework import serializers

from .models import Cat

class CatSerializer(serializers.ModelSerializer):

    class Meta:

        model = Cat

        fields = ('name', 'color', 'birth\_year')

Чтобы подключить сериализатор к обработке данных, во view-функции **cat\_list()** нужно создать экземпляр класса **CatSerializer** и передать в него данные из тела POST-запроса.

Данные в запросе приходят в формате JSON, преобразуются в Python-словарь, доступ к которому можно получить через объект **request.data**. Этот словарь и передаётся в сериализатор через именованный параметр **data**.

from .serializers import CatSerializer

@api\_view(['GET', 'POST'])

def cat\_list(request):

    # Обработчик для POST-запросов.

    if request.method == 'POST':

        serializer = CatSerializer(data=request.data)

**Валидация входящих данных**

Для валидации полученных данных надо вызвать в объекте сериализатора **serializer** метод **is\_valid()**. В зависимости от результатов валидации можно среагировать на запрос по-разному:

* если валидация прошла успешно — сохраним запись в БД при помощи метода **save()** и в качестве подтверждения вернём в ответе созданный объект и статус-код, соответствующий успешному выполнению операции;
* если валидация не пройдена — вернём в ответе объект **serializer.errors**. В этом объекте сериализатор автоматически создаёт словарь с перечнем ошибок, возникших при валидации. Вместе с перечнем ошибок вернём и статус-код, соответствующий неудачному результату выполнения операции.

При разработке Rest API хорошей практикой считается всегда возвращать соответствующий ситуации статус-код ответа. Если в коде не указать статус ответа — он всё равно будет отправлен автоматически; однако в таком случае может вернуться код, не отражающий реальное состояние дел.

Добавим в код валидацию данных. Теперь view-функция, которая добавляет в БД новую запись, будет выглядеть так:

@api\_view(['GET', 'POST'])

def cat\_list(request):

    if request.method == 'POST':

        # Создаём объект сериализатора

        # и передаём в него данные из POST-запроса

        serializer = CatSerializer(data=request.data)

        if serializer.is\_valid():

            # Если полученные данные валидны —

            # сохраняем данные в базу через save().

            serializer.save()

            # Возвращаем JSON со всеми данными нового объекта

            # и статус-код 201

            return Response(serializer.data, status=status.HTTP\_201\_CREATED)

        # Если данные не прошли валидацию —

        # возвращаем информацию об ошибках и соответствующий статус-код:

        return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

**View-функция API: обработка POST-запроса со списком объектов**

Следующая задача — научить view-функцию принимать и обрабатывать список объектов. Такой подход будет востребован: если владелец котофермы захочет добавить в базу всех своих котов, ему не придётся отправлять сотню отдельных запросов.

POST-запрос на добавление нескольких объектов будет выглядеть иначе, чем в случае с одним объектом: в теле POST-запроса будет передан не объект с полями **name**, **color** и **birth\_year**, а список таких объектов:

# Пример JSON со списком объектов

[

    {

        "name": "Стёпа",

        "color": "белый",

        "birth\_year": 1971

    },

    {

        "name": "Мурка",

        "color": "рыжий",

        "birth\_year": 2010

    },

    {

        "name": "Пушок",

        "color": "чёрный",

        "birth\_year": 2018

    }

]

Чтобы сериализатор был готов принять список объектов, в конструктор сериализатора нужно передать именованный параметр **many=True**.

serializer = CatSerializer(data=request.data, many=True)

Запрос со списком объектов будет обработан по тому же принципу, что и запрос с одним объектом.

Если этот параметр не указан, сериализатор не станет обрабатывать список объектов и вернёт ошибку: "*Invalid data. Expected a dictionary, but got list*."

Если в сериализаторе указан параметр **many=True**, а в запросе передан отдельный объект вместо списка объектов — сериализатор вернет ошибку: "*Expected a list of items but got type dict*."

**Обработка GET-запроса на получение списка объектов**

Вернуть список объектов в ответ на соответствующий GET-запрос можно с помощью той же **view-функции**.

При подготовке ответа view-функция должна получить из базы данных **queryset**, в котором будут храниться запрошенные объекты модели (например, все объекты модели **Cat**).

Затем queryset нужно сериализовать. Для этого queryset передаётся первым аргументом в конструктор сериализатора; вторым аргументом будет **many=True**:

# Получаем все объекты модели

cats = Cat.objects.all()

# Передаём queryset в конструктор сериализатора

serializer = CatSerializer(cats, many=True)

В результате view-функция API **cat\_list()** примет такой вид:

@api\_view(['GET', 'POST'])  # Разрешены только POST- и GET-запросы

def cat\_list(request):

    # В случае POST-запроса добавим список записей в БД

    if request.method == 'POST':

        serializer = CatSerializer(data=request.data, many=True)

        if serializer.is\_valid():

            serializer.save()

            return Response(serializer.data, status=status.HTTP\_201\_CREATED)

        return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

    # В случае GET-запроса возвращаем список всех котиков

    cats = Cat.objects.all()

    serializer = CatSerializer(cats, many=True)

    return Response(serializer.data)

Если при обработке GET-запроса в сериализаторе не указать параметр many=True, вернётся ошибка *AttributeError*.

**Создать/обновить. Сериализатор и метод PATCH**

В результате вызова метода **save()** сериализатора может быть создана новая запись в БД, а может быть обновлена существующая запись.

Для создания новой записи в сериализатор передаются только данные из запроса; объект будет создан при вызове метода **save()**:

...

serializer = CatSerializer(data=request.data)

# Если вызвать serializer.save(), будет создана новая запись в БД

Для обновления существующей записи первым параметром в сериализатор передаётся тот объект модели, который нужно обновить. В этом случае вызов **save()** не приведёт к созданию нового объекта.

...

cat= Cat.objects.get(id=id)

serializer = CatSerializer(cat, data=request.data)

# Если вызвать serializer.save(), будет обновлён существующий экземпляр Cat

В проекте **Kittygram** может возникнуть необходимость обновить существующие записи. Например, котик Анатолий 2018 года рождения был серый, но его перекрасили в рыжий. Надо обновить запись в базе данных, но потребуется заменить значение лишь одного поля, не перезаписывая весь объект целиком.

Положим, запись об Анатолии хранится в базе данных под *id=15*. Для частичного обновления данных следует отправить PATCH-запрос на эндпоинт *http://127.0.0.1:8000/cats/15/*; в теле этого запроса будет описание лишь одного поля:

{

  "color": "Рыжий"

}

По умолчанию сериализатор ожидает получить значения всех полей, перечисленных в его параметре **fields**. Если же сериализатор получит не все значения — это вызовет ошибку *400 Bad Request "This field is required."*: «*в запросе не передано обязательное поле!».*

Причиной такой ошибки может стать PATCH-запрос. Если при создании экземпляра сериализатора указать аргумент **partial=True** — отсутствие в запросе обязательных полей не приведёт к ошибке.

serializer = CatSerializer(cat, data=request.data, partial=True)

**Разделение обязанностей между view-функциями API**

Классический API для целевой модели — это, как правило, реализация шести операций:

* создание нового объекта;
* получение информации об объекте;
* удаление объекта;
* замещение объекта (целиком);
* изменение одного или нескольких полей объекта;
* получение списка объектов.

Хорошая практика — сгруппировать эти операции в две view-функции.

Первая view-функция добавляет новые объекты в коллекцию или возвращает все объекты коллекции, например:

* POST-запрос на адрес *cats/* создаст новую запись о котике;
* GET-запрос на тот же адрес *cats/* вернёт список всех котиков.

Вторая view-функция обрабатывает запросы для получения, изменения (полного или частичного) и удаления одиночного объекта:

* GET-запрос к адресу *cats/<pk>/* вернёт информацию о конкретном котике по его **id**;
* запросы PUT, PATCH или DELETE к тому же адресу *cats/<pk>/* перезапишут, изменят или удалят существующую запись о котике.

В результате для всех шести действий в **urls.py** потребуется описать лишь два эндпоинта:

* cats/,
* cats/<int:pk>/.

А ссылаться эти эндпоинты будут на две view-функции во **views.py**, например:

* cat\_list(),
* cat\_detail().

# 9 View-классы API

Следующий этап работы с Views в **Django REST framework** — встроенные **view-классы**. У них множество преимуществ перед view-функциями:

* возможность применять готовый код для решения стандартных задач;
* наследование, которое позволяет повторно использовать уже написанный код.

Встроенные классы DRF можно условно разделить на низкоуровневые и высокоуровневые. **Низкоуровневые** содержат лишь базовую структуру класса, его скелет; разработчик сам должен описать работу класса; их применяют для решения нестандартных задач.

Типовые задачи (скажем, CRUD) удобнее решать с помощью **высокоуровневых** view-классов: в них уже заготовлены все инструменты для решения стандартных задач.

**Низкоуровневые view-классы в DRF**

Начнём с низкоуровневого view-класса **APIView** из модуля **rest\_framework.views**.

Если view-класс унаследован от класса **APIView**, то при получении GET-запроса в классе будет вызван метод **get()**, а при получении POST-запроса — метод **post()**. Такие методы описаны для всех типов запросов, но по умолчанию эти методы не выполняют никаких действий, их нужно описывать самостоятельно.

# Скелет есть, а кода нет. Надо самостоятельно описать необходимые методы.

class MyAPIView(APIView):

    def get(self, request):

        ...

    def post(self, request):

        ...

    def put(self, request):

        ...

    def patch(self, request):

        ...

    def delete(self, request):

        ...

В целом этот класс работает так же, как и view-функции.

Измените проект Kittygram: вместо view-функции опишите view-класс.

Импортируйте класс **APIView** из **rest\_framework.views**, создайте класс-наследник и переопределите в нём методы **get()** и **post()**. Код почти ничем не будет отличаться от того, что был во view-функции, но будет написан в объектно-ориентированном стиле.

# Обновлённый views.py

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import status

from .models import Cat

from .serializers import CatSerializer

class APICat(APIView):

    def get(self, request):

        cats = Cat.objects.all()

        serializer = CatSerializer(cats, many=True)

        return Response(serializer.data)

    def post(self, request):

        serializer = CatSerializer(data=request.data)

        if serializer.is\_valid():

            serializer.save()

            return Response(serializer.data, status=status.HTTP\_201\_CREATED)

        return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

Чтобы всё заработало, исправьте код в **urls.py**, ведь синтаксис вызова view-классов отличается от синтаксиса вызова view-функций.

# urls.py

from django.urls import include, path

from cats.views import APICat

urlpatterns = [

    path('cats/', APICat.as\_view()),

]

Как и при работе со view-функциями, все операции CRUD при использовании view-классов принято разделять на 2 группы: в одном view-классе описывается создание нового объекта и запрос всех объектов (например класс **APICat**), а в другом классе — *получение/изменение/удаление* определённого объекта (например класс **APICatDetail**).

**Generic Views: высокоуровневые view-классы**

Для типовых действий, например, для вывода списка объектов или для запроса объекта по id удобнее использовать высокоуровневые view-классы, «**дженерики**» (англ. Generic Views): в них уже реализованы все механизмы, необходимые для решения задачи.

Некоторые из Generic Views выполняют строго определённую задачу (например, обрабатывают только один тип запросов), другие — более универсальны и могут «переключаться» на разные задачи в зависимости от HTTP-метода, которым был отправлен запрос.

В дженериках задают всего два поля: **queryset** (набор записей, который будет обрабатываться в классе) и **serializer\_class** (сериализатор, который будет преобразовывать объекты в формат JSON). В DRF все **Generic Views** объединены в модуле **rest\_framework.generics**.

Для работы возьмём два класса и на них реализуем все шесть операций классического API:

* комбинированный класс **ListCreateAPIView**: он возвращает всю коллекцию объектов (например, всех котиков) или может создать новую запись в БД;
* комбинированный класс **RetrieveUpdateDestroyAPIView**: его работа — возвращать, обновлять или удалять объекты модели по одному.

Импортируйте в код всё необходимое: **generics** из rest\_framework, **модель** и **сериализатор**; затем опишите дженерики:

# Обновлённый views.py

from rest\_framework import generics

from .models import Cat

from .serializers import CatSerializer

class CatList(generics.ListCreateAPIView):

    queryset = Cat.objects.all()

    serializer\_class = CatSerializer

class CatDetail(generics.RetrieveUpdateDestroyAPIView):

    queryset = Cat.objects.all()

    serializer\_class = CatSerializer

Измените вызов view-класса в **urls.py**:

# Обновлённый urls.py

from django.urls import include, path

from cats.views import CatList, CatDetail

urlpatterns = [

    path('cats/', CatList.as\_view()),

    path('cats/<int:pk>/', CatDetail.as\_view()),

]

Теперь проект Kittygram поддерживает весь **API CRUD** для модели **Cat** (а не только получение списка всех котиков и добавление нового котика). При минимальном количестве изменений в коде мы сделали API для модели Cat в Kittygram! Да и кода стало меньше; а где меньше кода, там меньше ошибок.

В коде явным образом не описана обработка разных типов запросов: всё происходит «под капотом» view-классов.

**Специализированные Generic Views**

В коде Kittygram view-классы унаследованы от комбинированных дженериков: они выполняют все операции CRUD. Для решения нашей задачи именно комбинированные дженерики подходят лучше всего. Но в некоторых случаях применение комбинированных view-классов будет избыточным или даже опасным.

Например, при создании API «только для чтения» лучше подключить специализированный **ListAPIView**, который выполняет ровно одно действие: выводит список объектов в ответ на GET-запрос. Это лучше и с точки зрения безопасности, и с точки зрения отсутствия избыточного кода.

Есть ещё несколько специализированных view-классов DRF:

* **RetrieveAPIView** — возвращает один объект (обрабатывает только GET-запросы);
* **CreateAPIView** — создаёт новый объект (обрабатывает только POST-запросы);
* **UpdateAPIView** — изменяет объект (обрабатывает только PUT- и PATCH-запросы);
* **DestroyAPIView** — удаляет объект (обрабатывает только DELETE-запросы).

Эти классы описываются в коде точно так же, как и комбинированные view-классы модуля **rest\_framework.generics**. ([удобная шпаргалка по классам DRF](https://www.cdrf.co/))

# 10 Вьюсеты и роутеры

На прошлом уроке вы реализовали все типовые операции CRUD, для этого потребовалось лишь два класса-дженерика и два эндпоинта. Шесть операций упакованы в два класса — неплохо, но можно лучше.

Всякий раз, когда снова потребуется реализовать CRUD, придётся снова и снова описывать тот же набор из аналогичных классов. А почему бы не описать всё в одном классе?

И когда у кого-то возник этот вопрос — появились **вьюсеты**.

Вьюсет (англ. viewset, «набор представлений») — это высокоуровневый view-класс, реализующий все операции CRUD; он может вернуть объект или список объектов, создать, изменить или удалить объекты.

Во вьюсеты встроена обработка разных типов запросов, работа с сериализаторами и моделями, фильтрация и пагинация результатов, возврат ошибок. Не нужно ничего придумывать: всё работает «из коробки».

В библиотеке rest\_framework есть несколько разных вьюсетов, они хранятся в пакете viewsets.

Начнём с самого популярного вьюсета - **ModelViewSet**.

**Универсальный ModelViewSet**

Класс **ModelViewSet** может выполнять любые операции CRUD с моделью. От разработчика не требуется описывать методы для чтения и записи данных для модели: эти операции уже реализованы.

В классе, наследующемся от **ModelViewSet**, обязательно должны быть описаны два поля:

* в поле **queryset** задаётся выборка объектов модели, с которой будет работать вьюсет;
* в поле **serializer\_class** указывается, какой сериализатор будет применён для валидации и сериализации.

В начале работы нужно импортировать пакет **viewsets** и создать класс, наследующийся от **ModelViewSet**.

# views.py

from rest\_framework import viewsets

from .models import Cat

from .serializers import CatSerializer

class CatViewSet(viewsets.ModelViewSet):

    queryset = Cat.objects.all()

    serializer\_class = CatSerializer

Всё - класс, который обработает все шесть типичных действий, готов.

Класс **ModelViewSet** предоставляет отличный набор инструментов для всех востребованных операций при работе с моделями. В большинстве стандартных ситуаций он будет работать «из коробки».

**Класс ReadOnlyModelViewSet: только чтение**

В пакете **rest\_framework.viewsets** есть похожий на **ModelViewSet**, но ограниченный в правах класс **ReadOnlyModelViewSet**. Он может только получать данные модели, а записывать и изменять — не может.

Этот класс полезен в ситуациях, когда требуется только выдавать данные по запросу, без возможности их изменить. В остальном **ReadOnlyModelViewSet** работает точно так же, как и **ModelViewSet**.

**Роутеры**

При работе с view-классами и дженериками каждый эндпоинт отдельно описывается в **urls.py**. Но для вьюсетов есть более удобный и экономичный инструмент — **роутеры** (англ. routers).

С помощью роутера для заданных вьюсетов создаются эндпоинты по маске адреса:

* *URL-префикс/* и
* *URL-префикс/<int:pk>*.

В DRF есть два стандартных роутера: **SimpleRouter** и **DefaultRouter**. Они очень похожи, начнём с первого.

Добавьте роутеры в Kittygram. В файл **urls.py** импортируйте класс **SimpleRouter** и создайте экземпляр этого класса.

# urls.py

from rest\_framework.routers import SimpleRouter

router = SimpleRouter()

Чтобы роутер создал необходимый набор эндпоинтов, необходимо вызвать его метод **register()** (говорят «зарегистрировать эндпоинты»). В качестве аргументов этот метод принимает URL-префикс и название вьюсета, для которого создаётся набор эндпоинтов.

router.register('cats', CatViewSet)

После регистрации надо включить новые эндпоинты в список **urlpatterns**: перечень эндпоинтов будет доступен в **router.urls**.

Создание эндпоинтов через **router** выглядит так:

# urls.py

from rest\_framework.routers import SimpleRouter

from django.urls import include, path

from cats.views import CatViewSet

# Создаётся роутер

router = SimpleRouter()

# Вызываем метод .register с нужными параметрами

router.register('cats', CatViewSet)

# В роутере можно зарегистрировать любое количество пар "URL, viewset":

# например

# router.register('owners', OwnerViewSet)

# Но нам это пока не нужно

urlpatterns = [

    # Все зарегистрированные в router пути доступны в router.urls

    # Включим их в головной urls.py

    path('', include(router.urls)),

]

Только что созданный роутер сгенерирует два эндпоинта:

* *cats/*,
* *cats/<int:pk>/*.

Теперь через эти эндпоинты будут доступны любые операции с моделью:

* POST-запрос на *cats/* создаст новую запись.
* Запросы PUT, PATCH или DELETE к адресу *cats/<pk>/* изменят или удалят существующую запись.
* GET-запрос на те же адреса вернёт список объектов или один объект.

**Параметр name в эндпоинтах**

В **urls.py** для каждого маршрута можно указать необязательный параметр **name**. Если этот параметр определён, то во view-функциях или view-классах через функцию **reverse()** можно получить соответствующие URL — это очень удобно и соответствует принципу DRY.

Роутеры сами автоматически создают **name** для каждого эндпоинта, его значение создаётся

* из имени модели, с которой работает вьюсет,
* и суффикса:
  + *-list* (для эндпоинта, работающего с коллекцией объектов)
  + или *-detail* (для эндпоинта, работающего отдельным объектом).

Например, для роутера, созданного для вьюсета **CatViewSet().**

class CatViewSet(viewsets.ModelViewSet):

    queryset = Cat.objects.all()

    serializer\_class = CatSerializer

имена эндпоинтов будут такими:

urlpatterns = [

    # Здесь имя "cat" взято из queryset,

    # с которым работает вьюсет CatViewSet

    path('cat/', ..., name='cat-list'),

    path('cat/<int:pk>/', ..., name='cat-detail'),

]

При работе с вьюсетами и роутерами вы можете столкнуться с ошибкой «*не определён аргумент basename*»:

*'basename' argument not specified, and could not automatically determine the name from the viewset, as it does not have a '.queryset' attribute.*

Речь идёт о необязательном аргументе роутера **basename**: в нём можно вручную указать префикс для параметра **name** в эндпоинтах, созданных роутером.

Например, можно переопределить префикс cat в **name='cat-list'** и **name='cat-detail'**:

router.register('cats', CatViewSet, basename='tiger')

В результате **name** для эндпоинтов будут начинаться с **tiger**:

urlpatterns = [

    path('cat/', ..., name='tiger-list'),

    path('cat/<int:pk>/', ..., name='tiger-detail'),

]

Однако есть случаи, когда параметр **basename** **обязательно** должен быть указан. Это необходимо в тех случаях, когда **queryset** однозначно не задан во вьюсете, а определён через метод **get\_queryset()**.

# Если бы пользователи могли оставлять комментарии к котикам,

# то эндпоинт для работы с комментариями выглядел бы примерно так:

# cats/{cat\_id}/comments/

class CommentViewSet(viewsets.ModelViewSet):

    serializer\_class = CommentSerializer

    # queryset во вьюсете не указываем

    # Нам тут нужны не все комментарии, а только связанные с котом с id=cat\_id

    # Поэтому нужно переопределить метод get\_queryset и применить фильтр

    def get\_queryset(self):

        # Получаем id котика из эндпоинта

        cat\_id = self.kwargs.get("cat\_id")

        # И отбираем только нужные комментарии

        new\_queryset = Comment.objects.filter(cat=cat\_id)

        return new\_queryset

В подобных ситуациях создать **name** автоматически не получится, и параметр **basename** придётся указать явным образом.

**SimpleRouter vs DefaultRouter**

**DefaultRouter** — это расширенная версия **SimpleRouter**: он умеет всё то же, что и SimpleRouter, а в дополнение ко всему генерирует **корневой эндпоинт** */*, GET-запрос к которому вернёт список ссылок на все ресурсы, доcтупные в API.

Если применён **DefaultRouter**, то в ответ на GET-запрос к адресу *http://127.0.0.1:8000/* вернётся список ссылок на доступные ресурсы.

# 11 Сериализаторы для связанных моделей

В учебном проекте Kittygram описана лишь простая модель **Cat**; сериализаторы для таких моделей тоже довольно просты.

Но в реальных проектах моделей больше, они сложнее и практически всегда связаны друг с другом. Для таких структур придётся настраивать сериализаторы и вьюсеты более детально, забираясь им «под капот».

Начнём с сериализаторов: разберёмся со связанными и вложенными сериализаторами, узнаем, как «на лету» модифицировать данные в ответе.

**Приручение котиков**

Пора немного улучшить проект Kittygram в части бизнес-логики. Пусть у каждого котика будет владелец (**owner**).

Для начала создайте модель, в которой будут храниться данные о хозяевах-котоводах (имени и фамилии владельца будет достаточно):

Добавьте в модель **Cat** поле **owner**, оно будет связано с моделью Owner.

Связи в таблицах бывают разного типа:

* один-к-одному (**OneToOne**);
* один-ко-многим (**ForeignKey**);
* многие-ко-многим (**ManyToMany**).

В нашем случае у каждого владельца может быть много котиков, но у каждого котика может быть только один владелец. Такая связь называется «**один-ко-многим**».

Свяжите модель **Cat** через поле **owner** с моделью **Owner**:

from django.db import models

class Owner(models.Model):

    first\_name = models.CharField(max\_length=128)

    last\_name = models.CharField(max\_length=128)

    def \_\_str\_\_(self):

        return f'{self.first\_name} {self.last\_name}'

class Cat(models.Model):

    name = models.CharField(max\_length=16)

    color = models.CharField(max\_length=16)

    birth\_year = models.IntegerField()

    # Новое поле в модели:

    owner = models.ForeignKey(

        Owner, related\_name='cats', on\_delete=models.CASCADE)

    def \_\_str\_\_(self):

        return self.name

**Миграции с приключениями**

После изменения моделей создайте миграции: *python manage.py makemigrations*. Но не всё так просто — при создании миграций возникнет вопрос:

You are trying to add a non-nullable field 'owner' to cat without a default; we can't do that (the database needs something to populate existing rows).

Please select a fix:

 1) Provide a one-off default now (will be set on all existing rows with a null value for this column)

 2) Quit, and let me add a default in models.py

Select an option:

Суть конфликта проста. При миграции в БД будет создано новое поле **owner**; если бы в базе были записи — это поле было бы создано и для них. Поле **owner — обязательное**, значит, в существующих записях оно должно быть заполнено. «*И чем же мы заполним поле owner, если в базе вдруг уже есть записи?*» — спрашивает Django.

В момент создания миграций Django не знает, есть ли в модели Cat записи, и задаёт этот вопрос на всякий случай: если записи есть — что тогда делать?

Выберите первый вариант ответа: «Если в модели Cat существуют записи — заполни в них поле owner одним и тем же значением!» Пусть этим значением будет, например, 1.

>>> 1

Migrations for 'cats':

  cats\migrations\0005\_auto\_20230420\_0842.py

    - Create model Owner

    - Alter field birth\_year on cat

    - Rename table for cat to (default)

    - Add field owner to cat

Прежде чем применить миграции *python manage.py migrate*, необходимо очистить все записи из таблицы Cat, поскольку записи в таблице Owner с id=1 нет (в ней пока вообще нет записей). Иначе вылетит SQL-ошибка:

django.db.utils.IntegrityError: ОШИБКА:  INSERT или UPDATE в таблице "cats\_cat" нарушает ограничение внешнего ключа "cats\_cat\_owner\_id\_1ad8adbe\_fk\_cats\_owner\_id"

DETAIL:  Ключ (owner\_id)=(1) отсутствует в таблице "cats\_owner".

Продолжаем работу: напишите сериализатор **OwnerSerializer** для модели **Owner**:

from rest\_framework import serializers

from .models import Cat, Owner

class CatSerializer(serializers.ModelSerializer):

    class Meta:

        model = Cat

        fields = ('id', 'name', 'color', 'birth\_year')

class OwnerSerializer(serializers.ModelSerializer):

    class Meta:

        model = Owner

        fields = ('first\_name', 'last\_name')

Напишите вьюсет **OwnerViewSet**:

from rest\_framework import viewsets

from .models import Cat, Owner

from .serializers import CatSerializer, OwnerSerializer

class CatViewSet(viewsets.ModelViewSet):

    queryset = Cat.objects.all()

    serializer\_class = CatSerializer

class OwnerViewSet(viewsets.ModelViewSet):

    queryset = Owner.objects.all()

    serializer\_class = OwnerSerializer

Зарегистрируйте через **роутер** эндпоинты для нового ресурса:

from rest\_framework.routers import SimpleRouter, DefaultRouter

from django.urls import include, path

from cats.views import CatViewSet, OwnerViewSet

# router = SimpleRouter()

router = DefaultRouter()

router.register('cats', CatViewSet)

router.register('owners', OwnerViewSet)

urlpatterns = [

    path('', include(router.urls)),

]

Запустите проект и добавьте одного хозяина.

**Сериализаторы для связанных моделей**

В модели **Owner** нет поля **cats**, но эта модель связана с моделью **Cat** через **related\_name 'cats'**. Сериализаторы могут работать с моделями, которые связаны друг с другом: можно указать имя **cats** в качестве поля сериализатора. Добавьте это поле в **OwnerSerializer**:

class OwnerSerializer(serializers.ModelSerializer):

    class Meta:

        model = Owner

        fields = ('first\_name', 'last\_name', 'cats')

В модели **Cat** поле **Owner** вы уже создали, его надо добавить в сериализатор **CatSerializer**:

class CatSerializer(serializers.ModelSerializer):

    class Meta:

        model = Cat

        fields = ('id', 'name', 'color', 'birth\_year', 'owner')

POST-запросами на эндпоинт cats/ добавьте в базу данных несколько котиков.

По умолчанию для связанных полей модели сериализатор будет использовать тип **PrimaryKeyRelatedField**; этот тип поля в сериализаторе оперирует первичными ключами (**id**) связанного объекта.

Поэтому в POST-запросе мы указывали именно **id** хозяина, а не его **first\_name** или **last\_name**. А при GET-запросе к эндпоинту **owners/** в поле **cats** для каждого хозяина будет возвращаться список id связанных с ним котиков.

Для начала неплохо, но совсем не информативно: по id про объект ничего не узнаешь. Нужно изменить дефолтное поведение сериализатора и вернуть вместо id связанного объекта какую-то другую информацию об объекте.

В классе **Cat** описан метод **\_\_str\_\_**: для строкового представления объектов модели Cat используется содержимое поля **name**. Настроим сериализатор так, чтобы вместо непонятного **id** возвращалось строковое представление объекта.

**Тип поля StringRelatedField**

В сериализаторе **OwnerSerializer** переопределите тип поля cats с дефолтного **PrimaryKeyRelatedField** на **StringRelatedField**.

Роль **StringRelatedField** — получить строковые представления связанных объектов и передать их в указанное поле вместо **id**.

class OwnerSerializer(serializers.ModelSerializer):

    cats = serializers.StringRelatedField(many=True, read\_only=True)

    class Meta:

        model = Owner

        fields = ('first\_name', 'last\_name', 'cats')

Обратите внимание, при указании типа поля были переданы аргументы **many=True** и **read\_only=True**.

* Для поля **cats** в модели **Owner** установлена связь «*один-ко-многим*» (у одного хозяина может быть много котиков), следовательно, полю **cats** в сериализаторе надо разрешить обработку списка объектов. Для этого в нём указан аргумент **many=True**.
* Поля с типом **StringRelatedField** не поддерживают операции записи, поэтому для них всегда должен быть указан параметр **read\_only=True**.

Теперь точно так же измените код сериализатора **CatSerializer**: переопределите поле **owner**. При запросе к эндпоинту *cats/* в ответе должен отображаться не id хозяина котика, а строковое представление объекта модели **Owner**; параметр **many=True** в этом поле не нужен, ведь у котика может быть только один хозяин.

class CatSerializer(serializers.ModelSerializer):

    owner = serializers.StringRelatedField(read\_only=True)

    class Meta:

        model = Cat

        fields = ('id', 'name', 'color', 'birth\_year', 'owner')

**Другие типы related-полей**

Помимо **PrimaryKeyRelatedField** и **StringRelatedField** в сериализаторах можно использовать и другие типы связанных полей. Загляните в шпаргалку, там перечислены самые популярные из них. А весь список можно традиционно найти в [документации](https://www.django-rest-framework.org/api-guide/relations/).

**Новые достижения - модель Achievement**

Создайте модель **Achievement** (англ. «достижение»). В ней будут храниться описания подвигов, которые так любят совершать все представители кошачьих

В модель **Cat** добавьте новое поле — **achievements** (англ. «достижения»), оно будет связано с моделью **Achievement** через вспомогательную модель **AchievementCat** — её тоже опишите в коде.

from django.db import models

class Achievement(models.Model):

    name = models.CharField(max\_length=64)

    def \_\_str\_\_(self):

        return self.name

class Owner(models.Model):

    first\_name = models.CharField(max\_length=128)

    last\_name = models.CharField(max\_length=128)

    def \_\_str\_\_(self):

        return f'{self.first\_name} {self.last\_name}'

class Cat(models.Model):

    name = models.CharField(max\_length=16)

    color = models.CharField(max\_length=16)

    birth\_year = models.IntegerField(null=True)

    # Новое поле в модели:

    owner = models.ForeignKey(

        Owner, related\_name='cats', on\_delete=models.CASCADE)

    # Связь будет описана через вспомогательную модель AchievementCat

    achievements = models.ManyToManyField(Achievement, through='AchievementCat')

    def \_\_str\_\_(self):

        return self.name

# В этой модели будут связаны id котика и id его достижения

class AchievementCat(models.Model):

    achievement = models.ForeignKey(Achievement, on\_delete=models.CASCADE)

    cat = models.ForeignKey(Cat, on\_delete=models.CASCADE)

    def \_\_str\_\_(self):

        return f'{self.achievement} {self.cat}'

Каждое достижение может принадлежать любому количеству котиков, и каждый котик может обладать любым количеством достижений; это связь «**многие-ко-многим**».

После изменения моделей создайте и примените миграции.

**Вложенный сериализатор**

В работе со связанными моделями **Cat** и **Owner** мы получали лишь ссылку на связанный объект (это был его id или что-то, замещающее id), но сам объект так и оставался недоступен. Однако список достижений котика — это именно список связанных с котиком объектов из модели **Achievement**. Надо их как-то добыть.

Задача состоит в том, чтобы при запросе к эндпоинту **cats/** вместе с объектом котика вернуть список, состоящий из привязанных к этому зверю объектов.

Чтобы реализовать эту идею в сериализаторе, нужно вложить один сериализатор в другой: определить в сериализаторе поле, типом которого будет другой сериализатор. Таким образом вложенный сериализатор передаст в поле родительского сериализатора список объектов.

В нашем случае родительским сериализатором будет **CatSerializer**, а вложенным — **AchievementSerializer** (но сперва его надо написать).

class AchievementSerializer(serializers.ModelSerializer):

    class Meta:

        model = Achievement

        fields = ('id', 'name')

Теперь перенастроим **CatSerializer**: переопределим в нём поле **achievements**. По дефолту к этому полю в сериализаторе будет применён тип **PrimaryKeyRelatedField**. Но в наших планах было получить не **id** объектов **Achievement**, а его объекты целиком.

Назначьте типом поля **achievements** сериализатор **AchievementSerializer**:

class AchievementSerializer(serializers.ModelSerializer):

    class Meta:

        model = Achievement

        fields = ('id', 'name')

class CatSerializer(serializers.ModelSerializer):

    owner = serializers.StringRelatedField(read\_only=True)

    # Переопределяем поле achievements

    achievements = AchievementSerializer(read\_only=True, many=True)

    class Meta:

        model = Cat

        fields = ('id', 'name', 'color', 'birth\_year', 'owner', 'achievements')

Теперь поле **achievements** в **CatSerializer** будет получать объекты **Achievement**, сериализованные в **AchievementSerializer**. Достижений у каждого котика может быть много, значит, полю **achievements** нужно передать аргумент **many=True**. И пока ограничим доступ к полю — «*только для чтения*», с записью разберёмся позже.

**Операции записи с вложенными сериализаторами**

Сейчас для поля **owner** модели **Cat** явным образом указан тип **StringRelatedField**; запись в него невозможна. Но в это поле потребуется записывать данные, поэтому удалите из кода строку, где это поле переопределяется: это вернёт полю его тип «по умолчанию».

Вложенные сериализаторы по умолчанию доступны только для чтения. Поэтому установка параметра **read\_only=True** — не выход: нам нужно записывать достижения в БД. Значит, надо описать, как должны сохраняться данные.

При получении такого POST-запроса:

{

    "name": "Барсик",

    "color": "White",

    "birth\_year": 2017,

    "owner": 1,

    "achievements": [

        {"name": "поймал мышку"},

        {"name": "разбил вазу"}

    ]

}

порядок работы должен быть таким:

* из списка **serializer.validated\_data** извлечь и сохранить в переменную элемент **achievements**: в нём хранится список достижений котика;
* в базе данных создать запись о новом котике — для этого у нас есть вся необходимая информация; достижения котика в этом не участвуют — лежат в стороне, ждут обработки;
* перебрать полученный список достижений котика и сравнить каждое из достижений с имеющимися в базе данных записями:
* если проверяемый элемент уже есть в базе — в таблицу связей **AchievementCat** добавить связь этого достижения с новым котиком;
* если проверяемого элемента в базе нет — в базе достижений создать новую запись и в таблицу связей **AchievementCat** добавить связь этого достижения с новым котиком.
* вернуть **JSON** с объектом свежесозданного котика и списком его достижений.

Чтобы настроить сохранение данных, нужно переопределить метод **create()** в сериализаторе. Опишем его и укажем явным образом, какие записи в каких таблицах нужно создать:

class CatSerializer(serializers.ModelSerializer):

    # Переопределяем поле achievements

    # Убрали owner = serializers.StringRelatedField(read\_only=True)

    class Meta:

        model = Cat

        fields = ('id', 'name', 'color', 'birth\_year', 'owner', 'achievements')

    def create(self, validated\_data):

        # Уберем список достижений из словаря validated\_data и сохраним его

        achievements = validated\_data.pop('achievements')

        # Создадим нового котика пока без достижений, данных нам достаточно

        print(validated\_data)

        print(\*\*validated\_data)

        cat = Cat.objects.create(\*\*validated\_data)

        # Для каждого достижения из списка достижений

        for achievement in achievements:

            # Создадим новую запись или получим существующий экземпляр из БД

            current\_achievement, status = Achievement.objects.get\_or\_create(

                \*\*achievement)

            # Поместим ссылку на каждое достижение во вспомогательную таблицу

            # Не забыв указать к какому котику оно относится

            AchievementCat.objects.create(

                achievement=current\_achievement, cat=cat)

        return cat

**Исходные данные из запроса в сериализаторе**

При работе с сериализаторами бывает полезен доступ к тем данным, которые были переданы в сериализатор: например, если нужно проверить, было ли передано в запросе какое-нибудь необязательное поле. Эти данные хранятся в словаре **serializer.initial\_data**, и прямо сейчас они понадобятся.

Отправьте POST-запрос на добавления котика, но не указывайте список его достижений - вернётся ошибка: поле **achievements** является обязательным.

Если иное не определено на уровне **модели** или **сериализатора** **явным образом**, то все поля модели, перечисленные в сериализаторе, будут обязательными.

В модели **Cat** явным образом не указано, что поле **achievements** — необязательное. **Сериализатор** видит, что поле модели не описано как необязательное — и к собственному полю **achievements** применяет атрибут **required=True**.

Но поле **achievements** нельзя назначать обязательным: бывают котики и без достижений, и не должно быть запрета на добавление таких котиков в Kittygram. Модель **Cat** это позволяет, но сериализатор не даёт этого сделать. Следовательно, нужно явным образом определить в **сериализаторе** поле **achievements** как необязательное.

Установим для поля **achievements** в сериализаторе атрибут **required** со значением **False**:

class CatSerializer(serializers.ModelSerializer):

    achievements = AchievementSerializer(many=True, required=False)

…

Теперь сериализатор не будет беспокоиться, если этого поля нет в запросе, но возникнет другая ошибка - переопределённый метод **create()** в сериализаторе пытается сохранить данные из поля **achievements**, но поле теперь необязательное, данные в POST-запросе для сохранения не пришли — и всё сломалось.

Тут и понадобится словарь **initial\_data**: в методе **create()** проверим, пришло в запросе поле **achievements** или нет, и, в зависимости от результата, будем сохранять котика с достижениями или без них.

…

def create(self, validated\_data):

        # Если в исходном запросе не было поля achievements

        if 'achievements' not in self.initial\_data:

            # То создаём запись о котике без его достижений

            cat = Cat.objects.create(\*\*validated\_data)

            return cat

        # Уберем список достижений из словаря validated\_data и сохраним его

        achievements = validated\_data.pop('achievements')

        # Создадим нового котика пока без достижений, данных нам достаточно

        cat = Cat.objects.create(\*\*validated\_data)

        # Для каждого достижения из списка достижений

        for achievement in achievements:

            # Создадим новую запись или получим существующий экземпляр из БД

            current\_achievement, status = Achievement.objects.get\_or\_create(

                \*\*achievement)

            # Поместим ссылку на каждое достижение во вспомогательную таблицу

            # Не забыв указать к какому котику оно относится

            AchievementCat.objects.create(

                achievement=current\_achievement, cat=cat)

        return cat

# 12 Сериализаторы: дополнительные настройки

В проекте Kittygram информация о котике содержит и год его рождения. Человеку, знакомому с календарём и арифметикой, не составит труда вычислить возраст котика. Но пользователи кошачьего API попросили разработчиков доработать сервис и явно возвращать возраст их любимцев в ответе на запрос.

**Поле SerializerMethodField: обработка данных в сериализаторе**

**SerializerMethodField** — это поле для чтения, связанное с определённым методом, в котором будет вычислено значение этого поля. Метод, который будет вызван для поля *<имя\_поля>*, по умолчанию должен называться *get\_<имя\_поля>*.

Аргументами метода *get\_<имя\_поля>* должны быть **self** и сериализуемый объект (он передаётся во второй аргумент):

def get\_something(self, obj):

    ...

С помощью поля **SerializerMethodField** можно модифицировать существующее поле или создать новое.

Добавьте в сериализатор CatSerializer поле **age** (его нет в модели Cat); содержимое этого поля будет вычисляться «на лету» в методе **get\_age**:

...

import datetime as dt

...

class CatSerializer(serializers.ModelSerializer):

    achievements = AchievementSerializer(many=True)

    age = serializers.SerializerMethodField()

    class Meta:

        model = Cat

        fields = ('id', 'name', 'color', 'birth\_year', 'owner', 'achievements',

                  'age')

    def get\_age(self, obj):

        return dt.datetime.now().year - obj.birth\_year

Хорошей практикой считается не перегружать этот метод какими-либо тяжёлыми операциями.

**Пользовательский тип поля для сериализатора**

Разработчики фронтенда сообщили, что по просьбе пользователей сайта они разрабатывают виджет-палитру для выбора цвета котиков. Виджет будет передавать цвет на бэкенд через API в закодированном виде, в **hex-формате**, например — *#ff0000*.

Сейчас в БД для каждого котика хранится название его цвета; название цвета — это обычная строка. Сериализатор для модели тоже ожидает цвет в строковом формате. Если в POST-запросе в поле **color** вместо строки с названием цвета "*красный*" придёт строка с кодом цвета "*#ff0000*", то код будет записан в БД. А нам это не надо, это не консистентно: изначально было принято решение хранить цвета в человекочитаемом виде; не будем нарушать это правило.

На сегодняшний день в [спецификации цветов HTML и CSS](https://www.w3.org/wiki/CSS/Properties/color/keywords) определено и поименовано более сотни цветов.

Таким образом, задача сводится к тому, что необходимо создать такое поле для сериализатора, которое

* в режиме записи конвертирует код цвета в его название,
* в режиме чтения вернёт название цвета из БД.

Существует простая и удобная библиотека [webcolors](https://webcolors.readthedocs.io/en/1.11.1/index.html), которая позволяет конвертировать код цвета в его название и наоборот. Она нам пригодится: добавьте ее в *requirements.txt* и установите в виртуальное окружение.

Теперь разберёмся с полем сериализатора. Кроме стандартных, вложенных или related типов полей в сериализаторах можно описывать и применять собственные типы; это как раз то, что нужно.

Для создания собственного типа поля сериализатора нужно описать класс для нового типа, который будет унаследован от **serializers.Field** и описать в нём два метода: **def to\_representation(self, value)** (*для чтения*) и **def to\_internal\_value(self, data)** (*для записи*).

Опишем новый тип поля **Hex2NameColor** в **serializers.py**:

import webcolors

...

class Hex2NameColor(serializers.Field):

    # При чтении данных ничего не меняем - просто возвращаем как есть

    def to\_representation(self, value):

        return value

    # При записи код цвета конвертируется в его название

    def to\_internal\_value(self, data):

        # Доверяй, но проверяй

        try:

            # Если имя цвета существует, то конвертируем код в название

            data = webcolors.hex\_to\_name(data)

        except ValueError:

            # Иначе возвращаем ошибку

            raise serializers.ValidationError('Для этого цвета нет имени')

        # Возвращаем данные в новом формате

        return data

...

Теперь новый тип поля **Hex2NameColor** можно присвоить полю **color** в **CatSerializer**:

...

class CatSerializer(serializers.ModelSerializer):

    achievements = AchievementSerializer(many=True, required=False)

    age = serializers.SerializerMethodField()

    color = Hex2NameColor()  # Вот он - наш собственный тип поля

    class Meta:

        model = Cat

        fields = ('id', 'name', 'color', 'birth\_year', 'owner', 'achievements',

                  'age')

...

При отправке POST-запроса на добавление нового котика можно передать в качестве цвета, например, код *#228B22*, который будет сконвертирован и сохранен в БД как *forestgreen*. При этом существующие записи при GET-запросе будут возвращаться без изменений. Если в POST-запросе передать код цвета, для которого не существует названия — сериализатор вернёт ошибку для этого поля.

**Переименование полей: параметр source**

В модели **Achievement** есть поле **name**, но и в других моделях есть поля с таким именем. Разобраться можно, но в ответах это имя лучше поменять на более информативное **achievement\_name**.

Сериализатор, унаследованный от **ModelSerializer**, по умолчанию использует те же названия полей, что и в модели, с которой он работает. Эти же имена служат ключами в ответе API.

Необходимость изменить имена возникает достаточно часто. Эту задачу решают через переопределение поля и применение параметра **source**:

new\_field\_name = serializers.CharField(source='old\_field\_name')

Определите новое поле **achievement\_name** в сериализаторе, в качестве параметра передайте аргумент *source=<'оригинальное имя поля в модели'>.*

class AchievementSerializer(serializers.ModelSerializer):

    achievement\_name = serializers.CharField(source='name')

    class Meta:

        model = Achievement

        fields = ('id', 'achievement\_name')

Этот подход работает при использовании сериализатора как для чтения, так и для записи.

**Ограничение возможных значений поля: выбор из списка**

Изменим возможности для выбора цвета котиков: набор кошачьих расцветок не особо велик, и можно заранее описать все доступные цвета. А потом можно будет организовать категоризацию по цветам.

Это можно сделать на уровне моделей:

    from django.db import models

CHOICES = (

        ('Gray', 'Серый'),

        ('Black', 'Чёрный'),

        ('White', 'Белый'),

        ('Ginger', 'Рыжий'),

        ('Mixed', 'Смешанный'),

    )

...

class Cat(models.Model):

    name = models.CharField(max\_length=16)

    color = models.CharField(max\_length=16, choices=CHOICES)

    birth\_year = models.IntegerField()

    owner = models.ForeignKey(

        Owner, related\_name='cats', on\_delete=models.CASCADE, blank=True,

        null=True)

    achievements = models.ManyToManyField(Achievement, through='AchievementCat')

    def \_\_str\_\_(self):

        return self.name

...

То же можно сделать и на уровне сериализатора, указав для поля **color** тип **ChoiceField** и передав в параметр **choices** список с возможными вариантами:

...

from .models import Cat, Owner, Achievement, AchievementCat, CHOICES

...

class CatSerializer(serializers.ModelSerializer):

    achievements = AchievementSerializer(many=True)

    age = serializers.SerializerMethodField()

    # Теперь поле примет только значение, упомянутое в списке CHOICES

    color = serializers.ChoiceField(choices=CHOICES)

    class Meta:

        model = Cat

        fields = ('id', 'name', 'color', 'birth\_year', 'owner', 'achievements',

                  'age')

...

Изменили модели — не забыли создать и применить миграции. После этого можно проверять.

Теперь работа с цветом стандартизована; запрос, содержащий в поле **color** название любого неучтённого в списке цвета, вызовет ошибку *400 Bad Request*.

# 13 Регулярные выражения

**Регулярные выражения** (англ. *regular expressions* или *regExp*) — это самостоятельный язык, предназначенный для поиска строк и проверки их на соответствие какому-то шаблону.

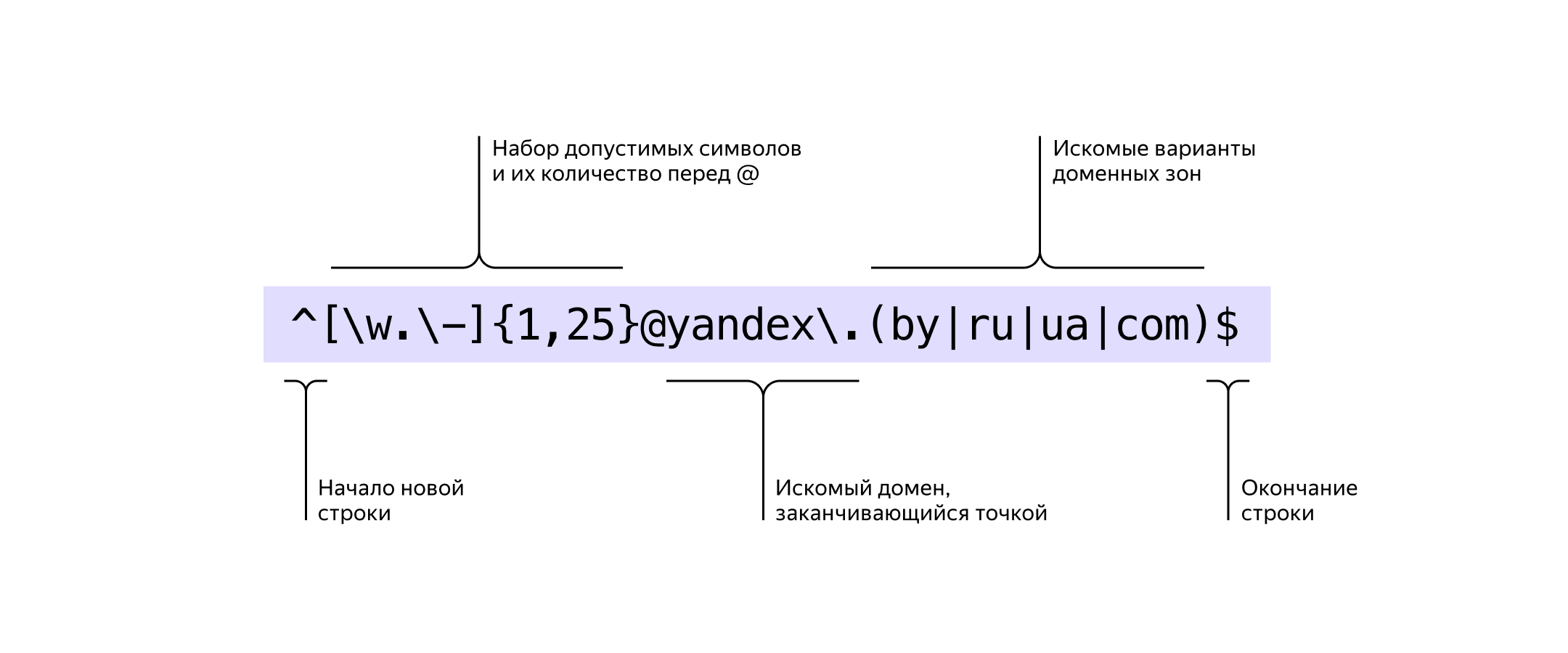
Для поиска используется строка-образец (англ. pattern, по-русски её часто называют «шаблоном» или «маской»), определяющая правило поиска.

Символы в регулярном выражении могут:

* служить спецсимволами шаблона (например, точка в шаблоне regExp означает «*любой символ в искомой строке*»),
* указывать на символы, которые должны присутствовать в искомой строке (в **email** перед названием доменной зоны обязательно должна стоять точка, это один из признаков **email**).

Чтобы отличить «*точку-спецсимвол regExp*» от «*точки — части искомой строки*», во втором случае символ «экранируют» — ставят перед ним обратный слеш.

Например, шаблон для поиска адреса электронной почты в доменах *yandex.ru, yandex.ua, yandex.by, yandex.com* получился таким:

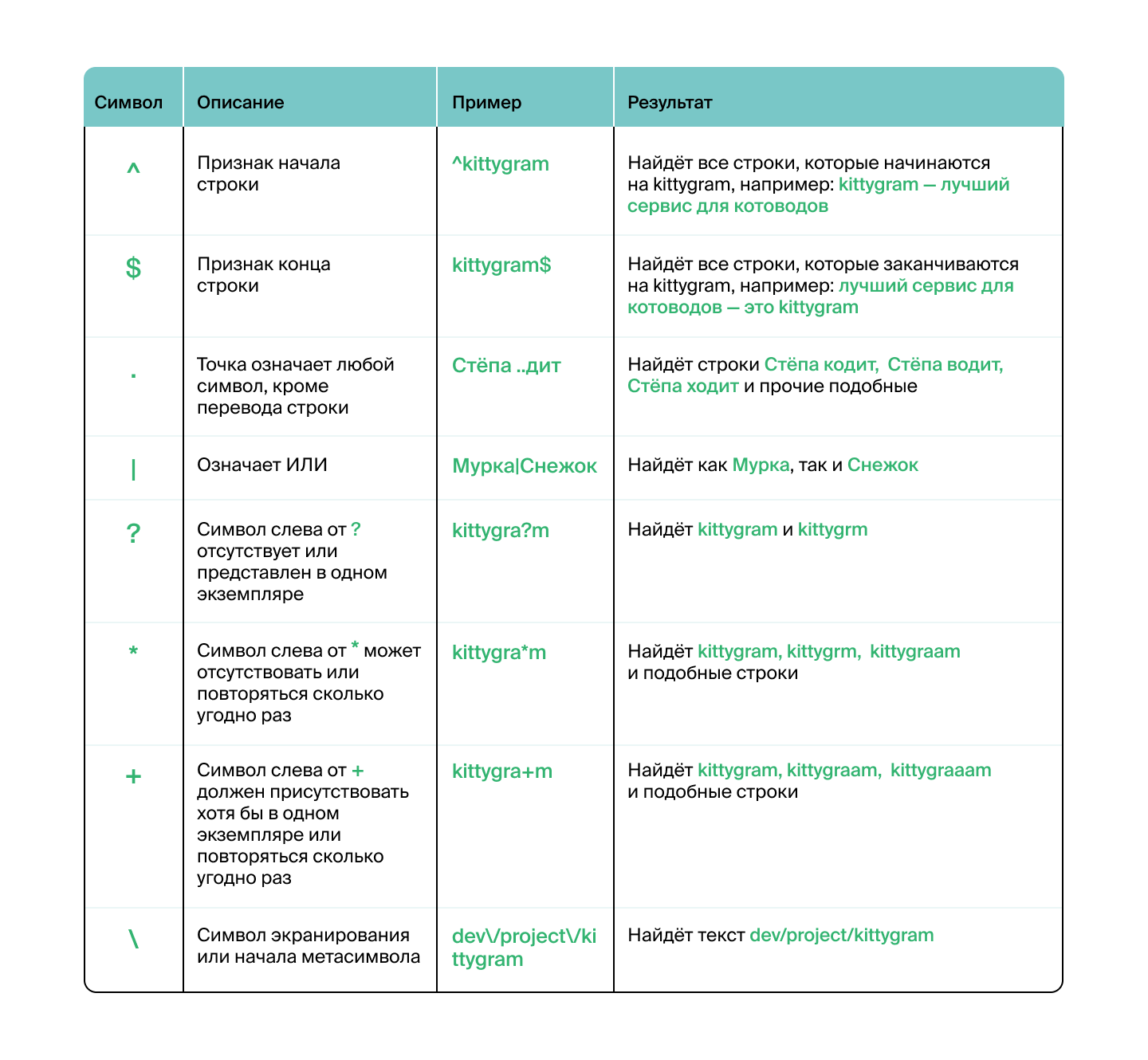


* **^** означает начало новой строки.
* **[\w.\-]** — квадратные скобки объединяют набор возможных символов, которые могут находиться в начале строки. Среди этих символов могут быть:
  + **\w** — любые словообразующие символы: буквы от a до z и от A до Z, цифры от 0 до 9, знак подчёркивания;
  + **.** — символ «точка» (в квадратных скобках её не нужно экранировать);
  + **\-** — символ «дефис» (экранирован, значит, это символ, который может встретиться в строке).

Эти символы используются чаще всего в первой части адреса электронной почты.

* **{1,25}** указывает на то, что строка, состоящая из символов, перечисленных в квадратных скобках перед фигурными скобками, может быть длиной от 1 до 25 знаков.
* Элемент **(…)** объединяет варианты доменов, которые могут быть в строке, а разделяющий их символ **|** соответствует оператору «или».
* **$** означает окончание строки.

Специальные символы и примеры их использования:



Помимо специальных символов есть метасимволы, которые описывают группы символов в строке, например:



Полный перечень специальных символов и метасимволов можно посмотреть в [документации Python](https://docs.python.org/3.9/library/re.html).

Полезные регулярные выражения для Django можно подсмотреть в шпаргалке, а потренироваться в написании можно в онлайн-тренажёре, например вот тут: <https://regex101.com/>.

Синтаксис регулярных выражений стандартизирован и применяется не только в Python, но и в большинстве других языков программирования. В них стоит разобраться: они пригодятся в работе не раз и не два.

# 14 Вьюсеты. Расширенные возможности

Бытует мнение, что у котиков с доминирующим белым окрасом голубой цвет глаз более глубокий, чем у остальных, и это считается достоинством. Вот именно они и представляют интерес для наших пользователей.

В связи с повышенным спросом на белых котиков требуется расширить возможности API: по специальному запросу нужно отдавать информацию о пяти последних добавленных котиках белого цвета. Для таких запросов уже придумали эндпоинт*: cats/recent-white-cats/*.

Задача не выглядит типичной: можно получить данные о всех котиках или о каком-то одном, но не о пяти, да ещё и определённого цвета.

**Нестандартные действия во вьюсетах**

Ещё раз переберём список стандартных действий (англ. actions) во вьюсетах:

* **create**: создание экземпляра;
* **retrieve**: получение экземпляра;
* **list**: получение списка экземпляров;
* **update**: обновление экземпляра (все поля);
* **partial\_update**: обновление экземпляра (только выбранные поля);
* **destroy**: удаление экземпляра.

Необходимо нестандартное действие; для этого во вьюсете пишут отдельные методы, которые оборачивают в декоратор **@action** («*действие*»). Этот декоратор настраивает метод и создаёт эндпоинты для этих действий.

Декоратор **@action** по умолчанию отслеживает только GET-запрос. Но если передать в декоратор параметр **methods**, то можно разрешить и другие методы запросов.

@action(methods=['get', 'delete', ...]

В декораторе можно явным образом указать, должен ли метод работать с одним объектом или с коллекцией объектов. Для этого используется параметр **detail**, который может принимать значения **True** (разрешена работа с одним объектом) или **False** (работаем с коллекцией).

URL эндпоинта по умолчанию генерируется из двух частей: *<URL-префикс ресурса>/<название метода>/*.

# К такому методу можно обратиться через эндпоинт cats/cats\_for\_sale/

@action()

def cats\_for\_sale()

    ...

Если URL эндпоинта не должен совпадать с именем метода, URL можно изменить: для этого нужно передать в декоратор аргумент **url\_path** с необходимым значением.

Решим задачу с получением последних пяти белых котиков. Для этого опишем метод **recent\_white\_cats** для вьюсета **CatViewSet**.

from rest\_framework.decorators import action

from rest\_framework.response import Response

...

class CatViewSet(viewsets.ModelViewSet):

    queryset = Cat.objects.all()

    serializer\_class = CatSerializer

    # Пишем метод, а в декораторе разрешим работу со списком объектов

    # и переопределим URL на более презентабельный

    @action(detail=False, url\_path='recent-white-cats')

    def recent\_white\_cats(self, request):

        # Нужны только последние пять котиков белого цвета

        cats = Cat.objects.filter(color='White')[:5]

        # Передадим queryset cats сериализатору

        # и разрешим работу со списком объектов

        serializer = self.get\_serializer(cats, many=True)

        return Response(serializer.data)

**Различные сериализаторы для одного вьюсета**

В Kittygram для сериализации и десериализации каждой модели использовался лишь один сериализатор. Но если некоторые поля сериализатора доступны только для чтения, то по факту для чтения применяется один набор полей, а для записи — другой. Это равносильно использованию разных сериализаторов.

Да, так тоже можно: для модели можно описать несколько сериализаторов и использовать их, например, в зависимости от типа запроса.

Допустим, когда

* добавляется новый котик,
* запрашивается детальная информация о конкретном котике,
* обновляется информация о конкретном котике

будем обрабатывать все доступные поля модели, а вот если запрашивается список котиков, то необходимы только **id**, **имя** и **цвет**.

Опишем для этого ещё один сериализатор, который назовём **CatListSerializer**:

class CatListSerializer(serializers.ModelSerializer):

    color = serializers.ChoiceField(choices=CHOICES)

    class Meta:

        model = Cat

        fields = ('id', 'name', 'color')

Добавьте во вьюсет стандартный метод **get\_serializer\_class**: в нём можно определить, какой из доступных сериализаторов должен обрабатывать данные в зависимости от действия:

class CatViewSet(viewsets.ModelViewSet):

    queryset = Cat.objects.all()

    serializer\_class = CatSerializer

    ...

    def get\_serializer\_class(self):

        # Если запрошенное действие (action) — получение списка объектов ('list')

        if self.action == 'list':

            # ...то применяем CatListSerializer

            return CatListSerializer

        # А если запрошенное действие — не 'list', применяем CatSerializer

        return CatSerializer

**Миксины и ваш собственный базовый класс для вьюсетов**

Чаще всего мы наследовали вьюсеты от **ModelViewSet**. По умолчанию это даёт возможность обрабатывать шесть типичных действий:

* создание нового объекта;
* получение информации об одном объекте;
* удаление объекта;
* полное замещение существующего объекта;
* изменение одного или нескольких полей объекта;
* получение списка объектов.

Если требуется только получать данные из БД, можно унаследоваться от **ReadOnlyModelViewSet**. В этом случае доступный набор действий будет таким:

* получение информации об одном объекте;
* получение списка объектов.

А если требуется иной набор действий — например, нужно только создавать новый объект и получать информацию об одном объекте? Эта задача может быть решена с использованием **миксинов** (от англ. *mix in*, «*смешивать*»). **Миксины** – это классы-«детали», из которых можно быстро собрать практически любой необходимый базовый вьюсет.

Чтобы самостоятельно создать базовый вьюсет с особым набором действий — нужно унаследовать его от одного или нескольких миксинов с нужными действиями и, дополнительно, от базового класса **GenericViewSet**:

from rest\_framework import mixins

...

# Собираем вьюсет, который будет уметь изменять или удалять отдельный объект.

# А ничего больше он уметь не будет.

class UpdateDeleteViewSet(mixins.UpdateModelMixin, mixins.DestroyModelMixin,

                          viewsets.GenericViewSet):

    pass

В DRF есть пять предустановленных классов миксинов, они соответствуют пяти операциям с данными:

* **CreateModelMixin** — создать объект (для обработки запросов POST);
* **ListModelMixin** — вернуть список объектов (для обработки запросов GET);
* **RetrieveModelMixin** — вернуть объект (для обработки запросов GET);
* **UpdateModelMixin** — изменить объект (для обработки запросов PUT и PATCH);
* **DestroyModelMixin** — удалить объект (для обработки запросов DELETE).

Опишем собственный базовый класс вьюсета: он будет создавать экземпляр объекта и получать экземпляр объекта; назовём его **CreateRetrieveViewSet**.

# cats/views.py

...

from rest\_framework import mixins

...

class CreateRetrieveViewSet(mixins.CreateModelMixin, mixins.RetrieveModelMixin,

                            viewsets.GenericViewSet):

    # В теле класса никакой код не нужен! Пустячок, а приятно.

    pass

Базовый вьюсет готов, он получился не хуже, чем **ModelViewSet**. Теперь можно унаследоваться от этого базового класса.

Опишем вьюсет **LightCatViewSet**, унаследованный от **CreateRetrieveViewSet**.

class LightCatViewSet(CreateRetrieveViewSet):

    queryset = Cat.objects.all()

    serializer\_class = CatSerializer

Зарегистрируем этот вьюсет в роутере:

from cats.views import CatViewSet, LightCatViewSet, OwnerViewSet

...

router.register(r'mycats', LightCatViewSet)

Теперь при запросе на новый эндпоинт *mycats/* GET-запрос к конкретному котику отрабатывает на «отлично», а вот получить список котиков через этот эндпоинт нельзя.

**Мой вьюсет — мои правила: наследование от класса ViewSet**

В тех случаях, когда нужно получить больше контроля и возможностей что-то «подкрутить» во вьюсетах, можно наследоваться от базового класса **ViewSet**. Именно от него наследуется, например, класс **ModelViewSet**.

Класс **ViewSet**, в свою очередь, наследуется от **APIView**.

При работе с низкоуровневыми вьюсетами все нужные методы вам придётся описать самостоятельно.

В классе ViewSet есть шесть предопределённых методов:

* **list(self, request)** — для получения списка объектов из *queryset*;
* **create(self, request)** — для создания объекта в модели;
* **retrieve(self, request, pk=None)** — для получения определённого объекта из *queryset*;
* **update(self, request, pk=None)** — для перезаписи (полного обновления) определённого объекта из *queryset*;
* **partial\_update(self, request, pk=None)** — для частичного обновления объекта из *queryset*;
* **destroy(self, request, pk=None)** — для удаления одного из объектов *queryset*.

Чтобы применить любой из этих методов, нужно полностью описать его; в классе **ViewSet** эти методы объявлены, но не описаны. Например, в приложении нужно создать вьюсет, который будет получать сериализованный объект одного котика (методом **retrieve()**) и полный список всех котиков (методом **list()**). Это можно сделать так:

from rest\_framework import viewsets

from rest\_framework.response import Response

from django.shortcuts import get\_object\_or\_404

from .models import Cat

from .serializers import CatSerializer

class CatViewSet(viewsets.ViewSet):

    def list(self, request):

        queryset = Cat.objects.all()

        serializer = CatSerializer(queryset, many=True)

        return Response(serializer.data)

    def retrieve(self, request, pk=None):

        queryset = Cat.objects.all()

        cat = get\_object\_or\_404(queryset, pk=pk)

        serializer = CatSerializer(cat)

        return Response(serializer.data)

Похоже на то, как мы работали в **APIView**, с той разницей, что здесь присутствуют поля **queryset** и **serializer**.

Документацию по предустановленным методам класса **ViewSet** можно посмотреть на [официальном сайте проекта](https://www.django-rest-framework.org/api-guide/viewsets/#viewset-actions).

# 14 Аутентификация по токену. JWT + Djoser

**Аутентификация**

Аутентификация на веб-сайтах, как правило, устроена так: при входе на сайт вы один раз вводите логин и пароль, система проверяет ваши данные и, в соответствии с вашими правами, предоставляет вам доступ к сайту. При переходе по страницам этого сайта вам не приходится каждый раз заново вводить свои данные: система запоминает вас, «сохраняет ваше состояние».

Но для REST API такой вариант не подходит: отсутствие состояния (**stateless**) — это один из основных принципов REST. Этот принцип означает, что каждый запрос к серверу не должен быть связан с предыдущими запросами, как будто их и не было.

Если провести аналогию с веб-сайтами — REST предполагает, что при каждом переходе на новую страницу вам снова и снова придётся вводить логин и пароль.

**Аутентификация на основе токенов**

Аутентификация с помощью токенов отлично вписывается в архитектуру REST. Клиент отправляет токен, кодовую последовательность символов, в заголовке каждого запроса к серверу. Если в базе данных существует пользователь, которому выдавался этот токен, то запрос будет обработан. Если токен не соответствует ни одному пользователю — запрос будет отклонён.

При каждом последующем запросе к API клиент также должен передавать этот токен, и каждый раз будет проводиться проверка прав доступа.

## 14.1 Аутентификация через Authtoken

В актуальной версии Kittygram изменять данные может любой желающий. Это не лучший вариант.

Настроим проект так, чтобы отправлять запросы к сервису могли только аутентифицированные пользователи. Прикрутим к проекту механизм аутентификации по токену.

Порядок действий для пользователя будет таким:

* Пользователь создаёт на проекте свою учётную запись, для входа в систему у него будет логин и пароль.
* Клиент отправляет запрос на специальный эндпоинт и в запросе передаёт логин и пароль. Если в базе данных существует такой пользователь и пароль совпадает с сохранённым в базе — в ответ клиент получает токен.
* Теперь клиент может работать с API, но при каждом запросе он должен отправлять токен.

В DRF есть готовый модуль, который предоставляет возможность аутентификации по токенам прямо «из коробки»: **Authtoken**.

Чтобы подключить механизм получения токена и аутентификации по токену в Django REST framework, нужно выполнить несколько действий:

1. Подключить модуль **authtoken**: добавить строку *'rest\_framework.authtoken*' в список **INSTALLED\_APPS**

# settings.py

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    'rest\_framework',

    'rest\_framework.authtoken',

    'cats.apps.CatsConfig',

]

1. В **settings.py** в настройках **REST\_FRAMEWORK** объявить новый способ аутентификации **TokenAuthentication**.
2. Запретить доступ всем неаутентифицированным пользователям: ограничение доступа настраивается с помощью **пермишенов** (англ. *permissions*, "*разрешения*"). Чтобы запретить доступ без токена, нужно добавить значение **IsAuthenticated** для ключа **DEFAULT\_PERMISSION\_CLASSES**:

REST\_FRAMEWORK = {

    'DEFAULT\_PERMISSION\_CLASSES': [

        'rest\_framework.permissions.IsAuthenticated',

    ],

    'DEFAULT\_AUTHENTICATION\_CLASSES': [

        'rest\_framework.authentication.TokenAuthentication',

    ]

}

1. Выполнить миграции, чтобы в базе данных создались поля для работы и хранения токена: *python manage.py migrate*.
2. Добавить маршрут для получения токена. Подключать можно в любом **urls.py**, мы подключим в головном:

from rest\_framework.authtoken import views

urlpatterns = [

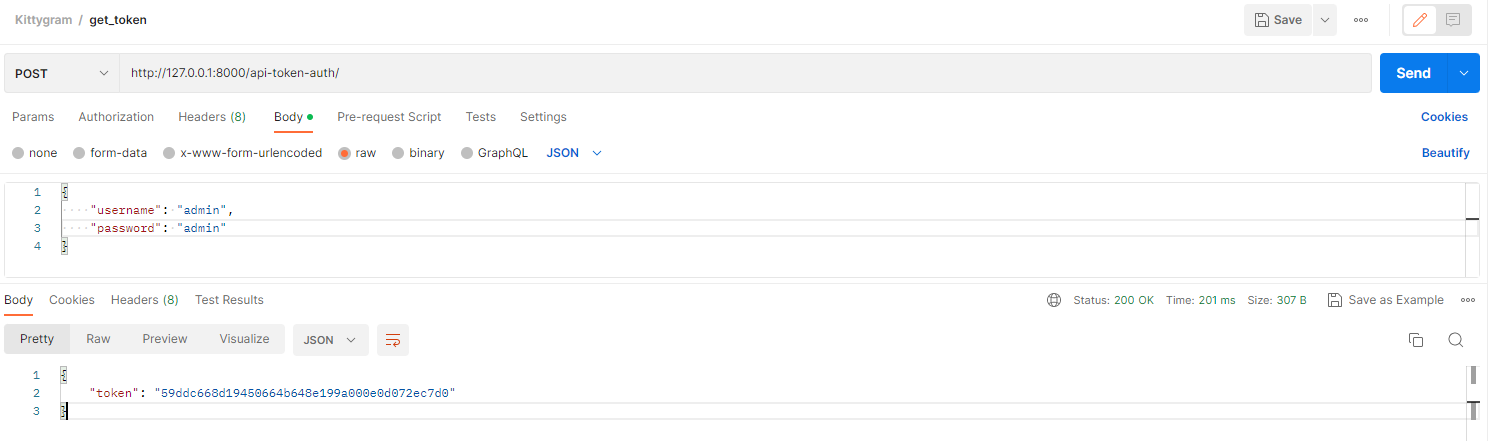
    ...

    path('api-token-auth/', views.obtain\_auth\_token),

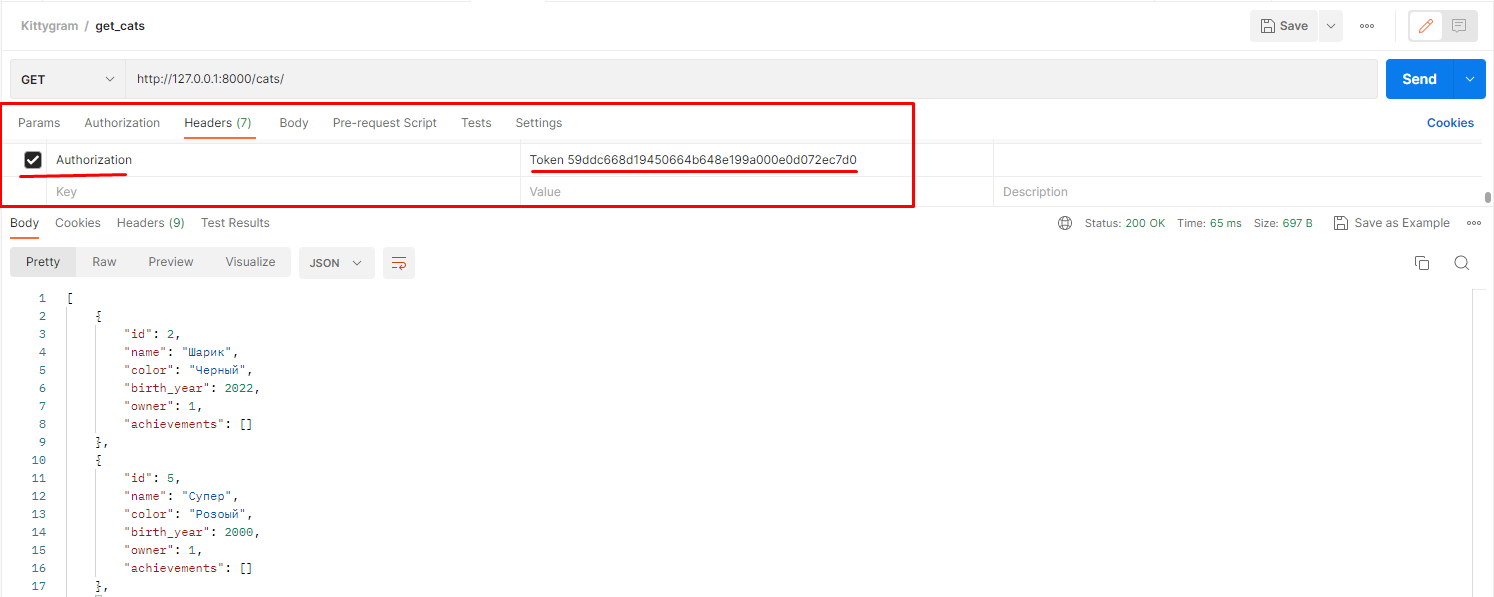
]

Дополнительную информацию о модуле **Authtoken** и о его настройках можно найти в [документации](https://www.django-rest-framework.org/api-guide/authentication/#tokenauthentication).

Теперь через Postman можно отправить POST-запрос к адресу *api-token-auth/*, передать в теле запроса **username** и **password** существующего пользователя (например, суперпользователя) — и в ответ придёт заветный токен:



Теперь этот токен надо будет передавать в заголовке каждого запроса, в поле **Authorization**. Перед самим токеном должно стоять ключевое слово **Token**.

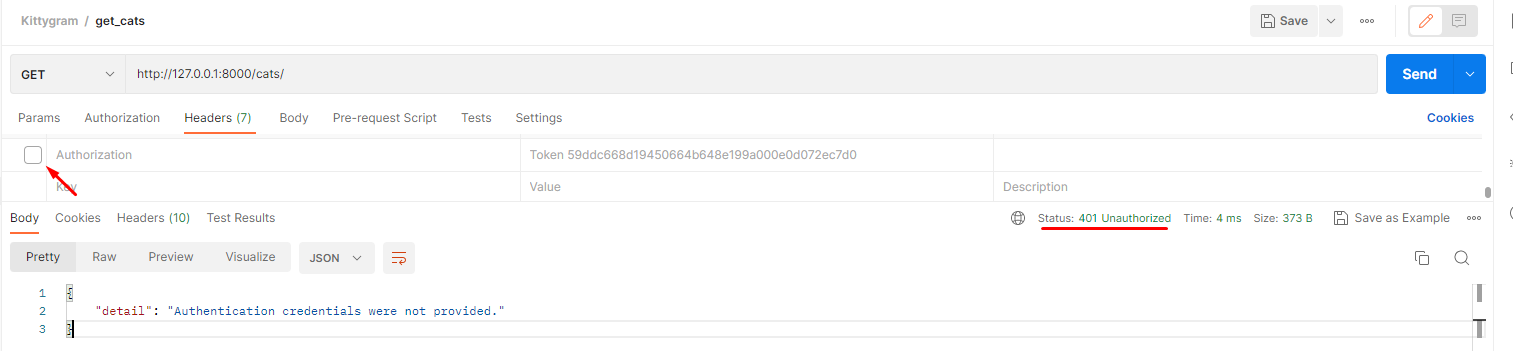


При последующих запросах к API после успешной аутентификации в объекте запроса будут доступны данные о пользователе:

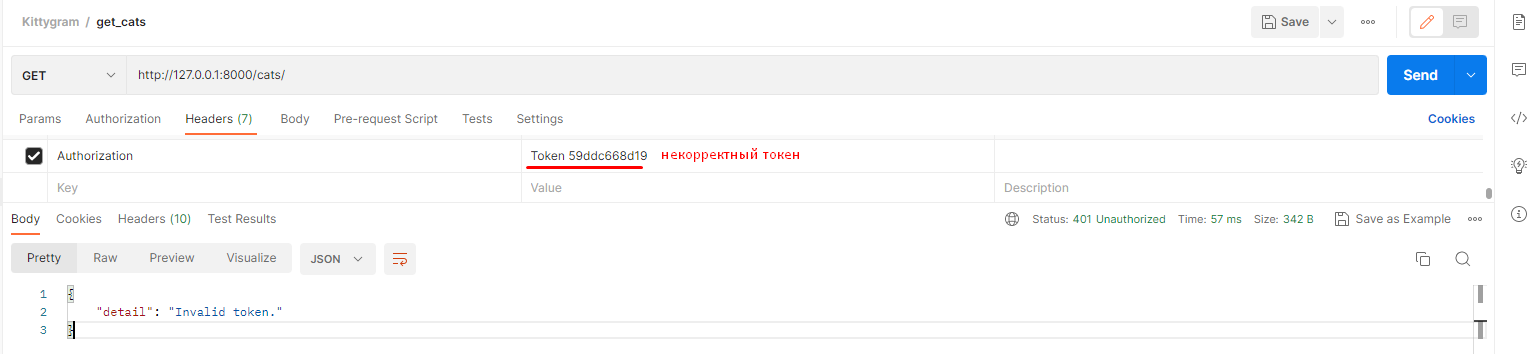
* **request.user** — экземпляр пользователя;
* **request.auth** — токен (экземпляр класса **rest\_framework.authtoken.models.Token**).

Этих данных не было в запросе, но **Authtoken** предоставляет их, основываясь на токене и соответствующей записи в базе данных. Их можно использовать, например, для того, чтобы сопоставить аутентифицированного пользователя с пользователем, который создавал соответствующий объект в БД.

В ответ на запрос без токена клиент получит отказ, статус-код **HTTP 401 Unauthorized**:



В случае указания недействительного токена (или токена с ошибкой) ответ будет таким:



## 14.2 Аутентификация по JWT-токену

Есть и более продвинутый способ аутентификации — по **JWT-токену**.

Преимущества JWT-токена в том, что прямо в нём записана информация о пользователе и сроке годности токена; системе не нужно каждый раз обращаться к базе данных, чтобы их сопоставить.

Токен, созданный по стандарту **JWT** (*JSON Web Token*), состоит из трёх частей. Каждая из них записывается в формате JSON:

* **header** (англ. «*заголовок*») содержит служебную информацию;
* **payload** (англ. «*полезная нагрузка*») хранит основные данные токена;
* **signature** (англ. «*подпись*») — подпись, ключ безопасности для защиты информации.

После подготовки каждая из частей кодируется алгоритмом **Base64URL**. Получившиеся строки разделяются между собой точками:



**Header JWT**

Header, как правило, содержит два поля:

* алгоритм создания подписи — обычно применяется алгоритм HMAC-SHA256 или RSA;
* тип токена — это строка "JWT".

{

"alg": "HS256",

"typ": "JWT"

}

**Payload JWT**

Payload хранит тип **токена**, **timestamp** со сроком его действия и информацию для аутентификации:

{

  "token\_type": "access",

  "exp":1578171903,

  "user\_id":5

}

**Signature JWT**

Подпись гарантирует, что содержимое **header** и **payload** в токене не было изменено после создания. Специальный алгоритм генерирует подпись на основе содержимого **header** и **payload**. При кодировании этот алгоритм использует секретный ключ, который известен только серверу.

С этого момента в учебном проекте Kittygram вы будете работать именно с **JWT-токенами**. Для этого придётся отключить аутентификацию через **Authtoken**.

**Настройка JWT-аутентификации**

Для работы с JWT в Django добавьте в *requirements.txt* установите две библиотеки [Djoser](https://djoser.readthedocs.io/en/latest/getting_started.html) и [Simple JWT](https://django-rest-framework-simplejwt.readthedocs.io/en/latest/):

*pip install djoser djangorestframework-simplejwt==4.7.2*

Обновите файл **settings.py**:

INSTALLED\_APPS = (

    'django.contrib.auth',

    ...

    'rest\_framework',

    'djoser',

)

Обратите внимание, приложение **Djoser** должно быть зарегистрировано после **django.contrib.auth** и **rest\_framework**.

Добавьте новые настройки в **settings.py**, они сходны с настройками Authtoken:

* permission;
* способ аутентификации по умолчанию;
* минимально необходимые настройки модуля Simple JWT.

from datetime import timedelta

...

REST\_FRAMEWORK = {

    'DEFAULT\_PERMISSION\_CLASSES': [

        'rest\_framework.permissions.IsAuthenticated',

    ],

    'DEFAULT\_AUTHENTICATION\_CLASSES': [

        'rest\_framework\_simplejwt.authentication.JWTAuthentication',

    ],

}

SIMPLE\_JWT = {

    # Устанавливаем срок жизни токена

   'ACCESS\_TOKEN\_LIFETIME': timedelta(days=1),

   'AUTH\_HEADER\_TYPES': ('Bearer',),

}

Проверьте, нет ли в проекте неприменённых миграций: *python manage.py migrate*.

Измените файл роутинга **urls.py**:

urlpatterns = [

    ...

    # Djoser создаст набор необходимых эндпоинтов.

    # базовые, для управления пользователями в Django:

    path('auth/', include('djoser.urls')),

    # JWT-эндпоинты, для управления JWT-токенами:

    path('auth/', include('djoser.urls.jwt')),

]

Готово. Но прежде чем продолжить — посмотрите на некоторые из эндпоинтов, которые стали доступны в API.



Хорошая новость заключается в том, что **djoser** сделал за вас большую часть тяжёлой работы: он сгенерировал эндпоинты, управляющие токенами и пользователями через API.

Список всех эндпоинтов, которые создаёт **djoser**, есть в [документации](https://djoser.readthedocs.io/en/latest/getting_started.html#available-endpoints).

Для обработки добавленных эндпоинтов djoser использует собственные вьюсеты и сериализаторы.

Если нужно изменить набор полей сериализатора из djoser, то

* из **djoser.serializers** импортируется класс сериализатора, который нужно переопределить (например, **UserSerializer** или **UserCreateSerializer**; полный список сериализаторов djoser доступен в [документации](Для%20обработки%20добавленных%20эндпоинтов%20djoser%20использует%20собственные%20вьюсеты%20и%20сериализаторы.));
* описывается новый класс сериализатора (он наследуется от импортированного);
* в новом сериализаторе переопределяется набор полей, используемых по умолчанию.

from djoser.serializers import UserSerializer

...

class CustomUserSerializer(UserSerializer):

    class Meta:

        model = User

        fields = ('email', 'id', 'username', 'first\_name', 'last\_name')

Аналогично можно поступить и с вьюсетом, который использовал сериализатор.

from djoser.views import UserViewSet

...

from .serializers import CustomUserSerializer

...

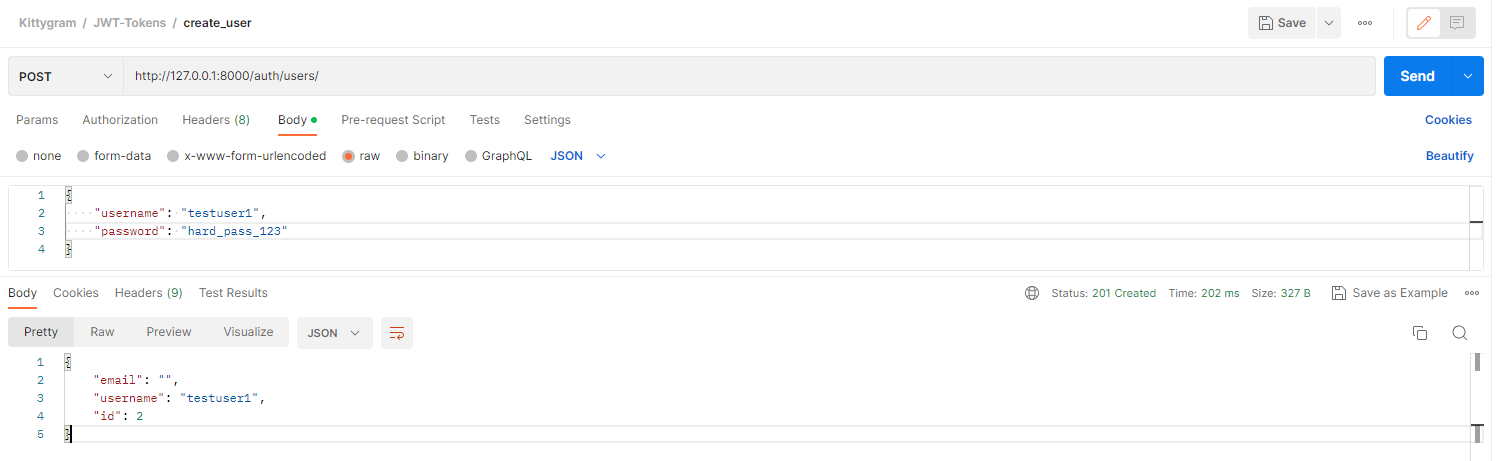
class CustomUserViewSet(UserViewSet):

    ...

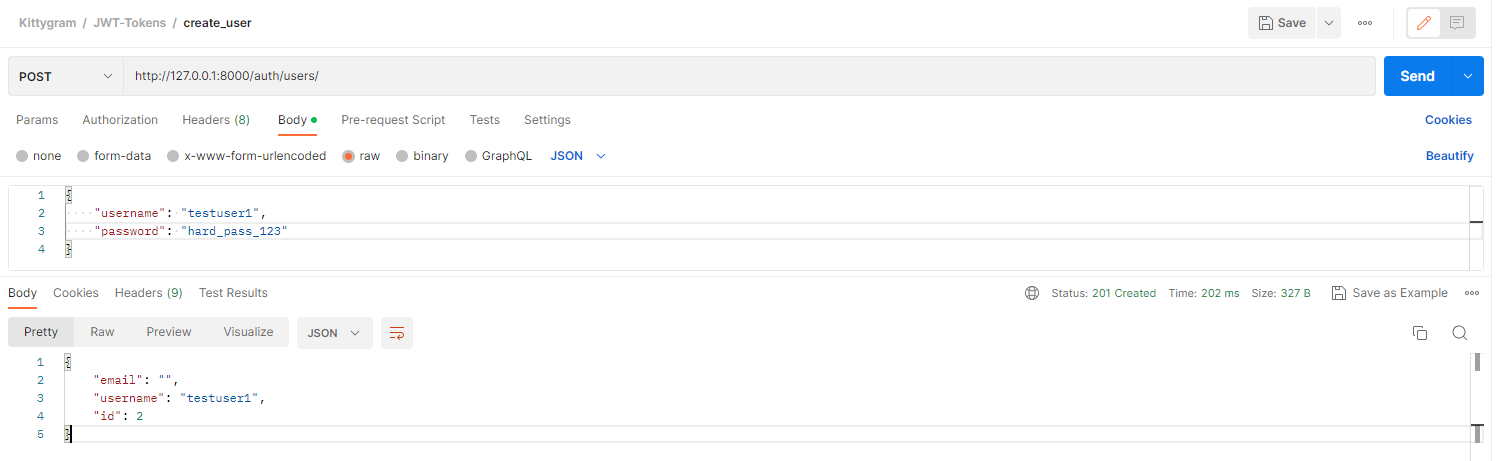
**Применение JWT на практике**

Теперь пользователя можно создать через API.

Придумайте новую пару «*логин-пароль*» и отправьте POST-запрос на *http://127.0.0.1:8000/auth/users/*, передав их в полях **username** и **password**.



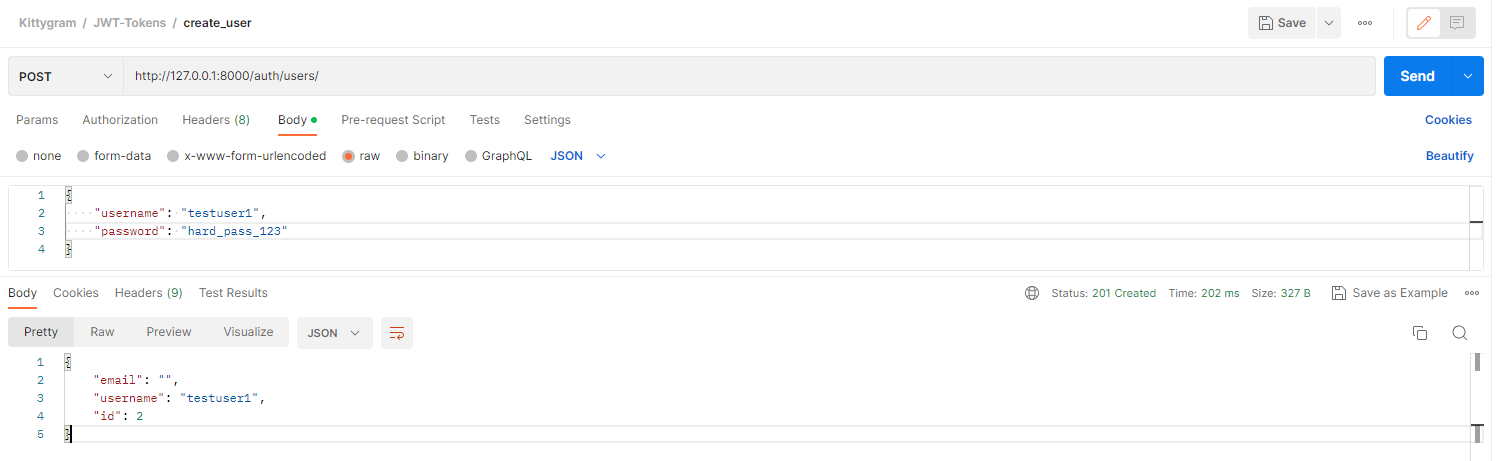
Теперь можно получить токен: отправьте POST-запрос на эндпоинт */auth/jwt/create/*, передав действующий логин и пароль в полях **username** и **password**.



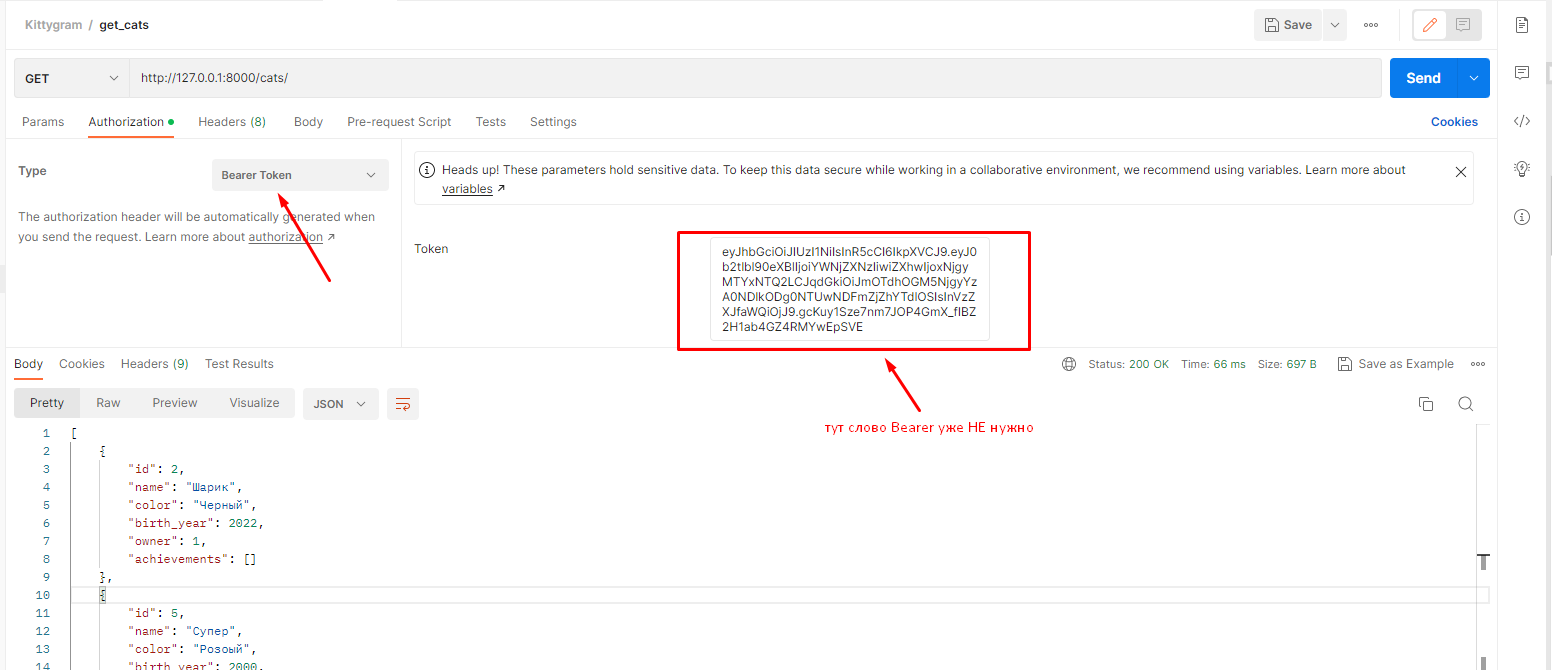
Токен вернётся в поле **access**, а данные из поля **refresh** пригодятся для обновления токена.

Если ваш токен утрачен, украден или каким-то иным образом скомпрометирован, вам понадобится отключить его и получить новый. Для этого отправьте POST-запрос на тот же адрес */auth/jwt/create/*, а в теле запроса в поле **refresh** передайте *refresh-токен*.

**Access** токен также надо будет передавать в заголовке каждого запроса, в поле **Authorization**. Перед самим токеном должно стоять ключевое слово **Bearer** и пробел. Слово **Bearer** (англ. «носитель») здесь заменяет слово **Token** и означает, что за ним следует сам токен.



В Postman существует возможность передавать токен и по-другому: можно выбрать соответствующий тип авторизации во вкладке **Authorization** и указать JWT-токен уже там.



# 15 Kittygram 2. Валидация

Склонируйте [новую версию проекта](https://github.com/anton8963kobelev/kittygram2), выполните миграции, запустите веб-сервер, создайте через API несколько пользователей и получите для них токены.

Что нового:

* Вместо модели **Owner** используем встроенную модель **User**;
* Аутентифицированные пользователи не должны иметь возможность удалять или изменять данные друг друга, поэтому вьюсет, отвечающий за работу с пользователями, унаследован от **ReadOnlyModelViewSet**.
* Чтобы дать возможность пользователям самостоятельно регистрироваться через API и обеспечить доступ по токену, воспользуемся связкой **JWT+Djoser**.

Выполните POST-запрос на эндпоинт cats/; анонимные запросы к проекту теперь запрещены, так что передайте с запросом токен одного из пользователей; в теле запроса передайте всю необходимую информацию о домашнем питомце:

{

    "name": "Барсик",

    "color": "Black",

    "birth\_year": 2020,

    "owner": 1

}

Если указать **id** другого пользователя (из списка существующих), то такой запрос тоже пройдёт. А вот это уже не очень хорошо, ведь добавлять, удалять и изменять информацию о котике может только хозяин. А пока что пользователь с *id=3* может добавить котика от имени пользователя с *id=1*.

Самый простой и очевидный способ исправить ситуацию — получать владельца котика не из поля **owner** в теле запроса, а из объекта **request**: там доступен экземпляр пользователя, которому принадлежит токен. А сериализатору надо сказать, чтобы он игнорировал это поле, если оно будет указано в теле POST-запроса.

Начнём с последнего пункта: переопределим поле **owner** и укажем параметр **read\_only**.

class CatSerializer(serializers.ModelSerializer):

    achievements = AchievementSerializer(read\_only=True, many=True)

    color = serializers.ChoiceField(choices=CHOICES)

    owner = serializers.PrimaryKeyRelatedField(read\_only=True)

    age = serializers.SerializerMethodField()

    class Meta:

        model = Cat

        fields = ('id', 'name', 'color', 'birth\_year','achievements', 'owner',

                  'age')

    ...

Параметр **read\_only\_fields** можно указать не в описании поля, а во внутреннем классе **Meta**: так удобнее — это позволяет перечислить все поля для чтения в одном месте, не переопределяя их.

class CatSerializer(serializers.ModelSerializer):

    achievements = AchievementSerializer(read\_only=True, many=True)

    color = serializers.ChoiceField(choices=CHOICES)

    age = serializers.SerializerMethodField()

    class Meta:

        model = Cat

        fields = ('id', 'name', 'color', 'birth\_year', 'achievements', 'owner',

                  'age')

        read\_only\_fields = ('owner',)

    ...

Теперь сериализатор не ждёт в теле POST-запроса поле **owner** (а если оно придёт, то будет проигнорировано).

**Передача собственного значения в метод save()**

Теперь вьюсет не может создать новую запись в БД, ведь поле **owner** в модели **Cat** обязательное, а данные из сериализатора в него не приходят: при POST-запросах сериализатор игнорирует поле **owner**.

Задача состоит в том, чтобы при создании новой записи в БД поле **owner** не осталось пустым, а в него была записана информация о пользователе, отправившем запрос. Эта информация у нас есть — она доступна в **request.user**, осталось в нужный момент подсунуть её в базу.

Новая запись в БД создаётся при вызове метода **save()** сериализатора, а этот метод вызывается из метода вьюсета **perform\_create()**.

Чтобы передать новое значение для какого-то поля в метод save(), нужно переопределить метод perform\_create().

В метод save() в полe **owner** передадим объект пользователя, отправившего запрос.

class CatViewSet(viewsets.ModelViewSet):

    queryset = Cat.objects.all()

    serializer\_class = CatSerializer

    def perform\_create(self, serializer):

        serializer.save(owner=self.request.user)

Вот теперь всё работает как надо: сериализатор не ожидает **id** пользователя в POST-запросе (или игнорирует его при получении), а при создании записи о новом котике в БД информация о пользователе берётся из объекта **request.user**.

При подобных операциях с PUT- и PATCH-запросами следует переопределить метод **perform\_update()**, а в остальном всё работает так же.

**Валидаторы**

По умолчанию валидация в сериализаторе происходит на базовом уровне. Если в поле ожидается целочисленное значение, а пришло число с плавающей точкой, то сериализатор вернёт ошибку.

А вот если в запросе придёт значение, которое по типу соответствует ожиданиям, но не удовлетворяет бизнес-логике или здравому смыслу (например, год рождения — в будущем), то ошибка не возникнет.

**Валидация на уровне поля**

Допустим, к проекту Kittygram отправлен POST-запрос, в котором переданы некорректные данные: например, год рождения котика указан в будущем или, наоборот, в далёком прошлом (а котики, к сожалению, не живут более сорока лет). Такие данные принимать нельзя, значит, надо провести валидацию полученных значений.

Проверку можно провести на уровне отдельно взятого поля. Сделаем это на примере поля **birth\_year**.

Метод для валидации определённого поля должен называться **validate\_<имя поля>**, он будет автоматически вызываться при получении данных.

Опишите в **CatSerializer** новый метод **validate\_birth\_year()**.

class CatSerializer(serializers.ModelSerializer):

    achievements = AchievementSerializer(many=True, required=False)

    color = serializers.ChoiceField(choices=CHOICES)

    age = serializers.SerializerMethodField()

    owner = serializers.PrimaryKeyRelatedField(read\_only=True)

    class Meta:

        model = Cat

        fields = ('id', 'name', 'color', 'birth\_year',

                      'achievements', 'owner', 'age')

    def validate\_birth\_year(self, value):

        year = dt.date.today().year

        if not (year - 40 < value <= year):

            raise serializers.ValidationError('Проверьте год рождения!')

        return value

Ещё один важный момент: этот метод не только проверяет данные, но и возвращает значение.

Дело здесь в том, что DRF позволяет не только проверять данные, но и **нормализовать** их, приводить к нужному виду — например, к верхнему или нижнему регистру. Здесь эта возможность не используется, но DRF требует вернуть значение, и поэтому функция возвращает то, что сама получила на вход, — аргумент **value**.

**Валидация на уровне объекта**

Метод **validate\_birth\_year** проверяет одно-единственное поле **birth\_year** и ничего не знает об остальных данных.

Однако значения отдельных полей могут по отдельности успешно проходить все валидации, а вот их совокупность не всегда бывает валидной.

Например, если в POST-запросе значения полей **name** и **color** совпали — это почти наверняка ошибка: редко бывает так, что имя котика совпадает с его цветом. Чтобы выявить эту ошибку, понадобится сравнить значения этих полей.

Чтобы выполнить проверку, требующую доступа к нескольким полям, добавьте в сериализатор метод **validate()**. Этот метод принимает один аргумент — словарь со значениями полей.

class CatSerializer(serializers.ModelSerializer):

    achievements = AchievementSerializer(many=True, required=False)

    color = serializers.ChoiceField(choices=CHOICES)

    age = serializers.SerializerMethodField()

    owner = serializers.PrimaryKeyRelatedField(read\_only=True)

    class Meta:

        model = Cat

        fields = ('id', 'name', 'color', 'birth\_year',

            'achievements', 'owner', 'age')

    ...

    def validate(self, data):

        if data['color'] == data['name']:

            raise serializers.ValidationError(

                'Имя не может совпадать с цветом!')

        return data

**Встроенные валидаторы для сериализаторов**

Для сериализаторов в DRF есть несколько встроенных классов-валидаторов, среди них есть **UniqueValidator** и **UniqueTogetherValidator**.

* **UniqueValidator** обеспечивает проверку уникальности значения поля.

В моделях такое ограничение описывается параметром **unique=True** для поля.

* **UniqueTogetherValidator** обеспечивает проверку уникальности комбинации (например, если проверка уникальности идёт по полям **name** и **color**, в базе никогда не окажется несколько записей с *name='Мурзик'* и *color='Black'*, пусть даже остальные поля будут у них различаться).

Разработчики Kittygram 2.0 не хотят, чтобы забывчивые хозяева котиков дублировали записи о своих любимцах в БД. Поэтому реализовали проверку уникальности записей о котиках на примере двух полей: owner и name. В базе данных не должно быть двух или более записей, у которых имя котика и хозяин совпадают.

Эти ограничения реализуются на уровне модели: они описываются в классе **Meta**. Например, через атрибут **unique\_together**:

class Cat(models.Model):

    name = models.CharField(max\_length=16)

    color = models.CharField(max\_length=16, choices=CHOICES)

    birth\_year = models.IntegerField()

    owner = models.ForeignKey(

        User, related\_name='cats', on\_delete=models.CASCADE)

    achievements = models.ManyToManyField(Achievement, through='AchievementCat')

    class Meta:

        unique\_together = ('name', 'owner')

    def \_\_str\_\_(self):

        return self.name

Но [документация Django](https://docs.djangoproject.com/en/3.2/ref/models/options/#unique-together) рекомендует вместо unique\_together использовать **UniqueConstraint**: этот способ обеспечивает большую функциональность, а unique\_together может быть признан устаревшим в будущем.

class Cat(models.Model):

    name = models.CharField(max\_length=16)

    color = models.CharField(max\_length=16, choices=CHOICES)

    birth\_year = models.IntegerField()

    owner = models.ForeignKey(

        User, related\_name='cats', on\_delete=models.CASCADE)

    achievements = models.ManyToManyField(Achievement, through='AchievementCat')

    class Meta:

        constraints = [

            models.UniqueConstraint(

                fields=['name', 'owner'],

                name='unique\_name\_owner'

            )

        ]

    def \_\_str\_\_(self):

        return self.name

Такую же проверку нужно реализовать и на уровне **сериализатора**.

В классе **Meta** сериализатора нужно указать опциональное поле **validators**, значением которого будет список валидаторов. Сейчас нужен только один валидатор: **UniqueTogetherValidator**.

У него есть два обязательных аргумента и один необязательный:

* **queryset**, обязательный: это queryset, для которого должна быть проверена уникальность.
* **fields**, обязательный: список или кортеж имён полей сериализатора, которые должны составлять уникальный набор.
* **message**: сообщение об ошибке на случай, если данные не прошли валидацию.

class CatSerializer(serializers.ModelSerializer):

    achievements = AchievementSerializer(many=True, required=False)

    color = serializers.ChoiceField(choices=CHOICES)

    age = serializers.SerializerMethodField()

    owner = serializers.PrimaryKeyRelatedField(read\_only=True)

    class Meta:

        model = Cat

        fields = ('id', 'name', 'color', 'birth\_year', 'achievements', 'owner',

                  'age')

        validators = [

            UniqueTogetherValidator(

                queryset=Cat.objects.all(),

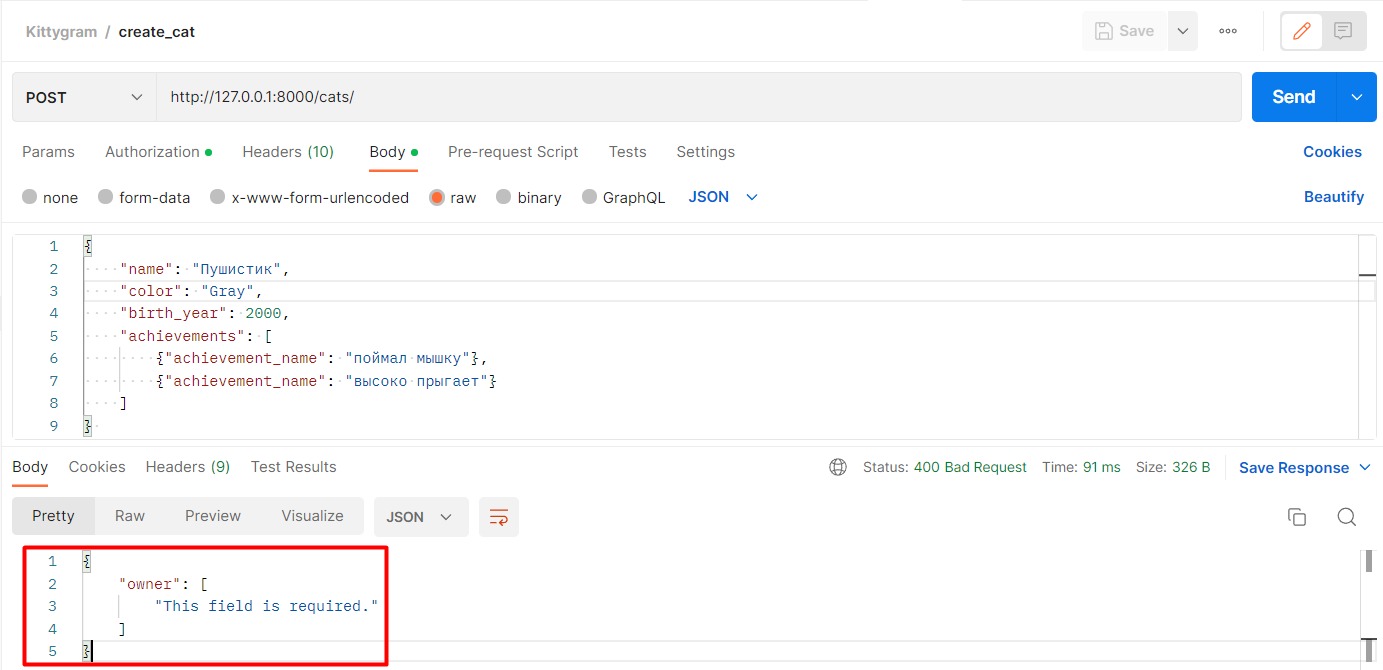
                fields=('name', 'owner')

            )

        ]

    ...

После изменений в модели создадим и применим миграции — и сделаем запрос на добавление нового котика. Начнём с такого, которого точно нет в БД.



Ошибка! Странное дело: DRF сообщает, что поле **owner — is required**, обязательное. Как так, мы же только что назначили его необязательным! Давайте разбираться что тут происходит.

Поле **owner** — необязательное, оно может отсутствовать в запросе (либо будет игнорироваться, если присутствует). Мы переопределили метод **save()**, теперь туда явным образом передаются данные для поля **owner**. Это происходит на этапе создания нового экземпляра класса.

Но этому этапу предшествует запуск валидаторов, и в этот момент сериализатор ещё не знает о новом содержимом поля **owner**: оно там просто отсутствует. Но валидатору это поле нужно: без него он не может выполнить проверку.

Выйти из сложившейся ситуации помогут дополнительные опции сериализатора: **скрытые поля** **и значения по умолчанию**.

**Скрытые поля и значения по умолчанию**

Для полей сериализатора можно указать значения по умолчанию. В этом случае, если в запросе будет отсутствовать требуемое поле, будет использовано заранее указанное значение.

Для поля **owner** можно указать значение по умолчанию, передав в него объект пользователя, отправившего запрос. Для определения пользователя есть встроенный класс **CurrentUserDefault** — именно его и нужно указать в качестве значения по умолчанию для поля **owner**.

Существует два варианта использования значений по умолчанию в сериализаторе для решения возникшей задачи.

1. Можно переопределить поле **owner**, указав ему тип **HiddenField** (скрытое поле), и передать ему значение по умолчанию **CurrentUserDefault**.

**HiddenField** не принимает передаваемые в запросе данные и всегда возвращает значение по умолчанию в словарь **validated\_data** в сериализаторе.

class CatSerializer(serializers.ModelSerializer):

    ...

    owner = serializers.HiddenField(default=serializers.CurrentUserDefault())

    ...

В этом случае поле **owner** всегда будет присутствовать в словаре **validated\_data** (и больше нет необходимости передавать его значение в метод **save()** во вьюсете), но его нельзя будет использовать в ответах сериализатора, даже если это поле будет явно указано в списке необходимых полей: оно скрытое, такие поля не попадают в ответ.

1. Можно переопределить поле **owner**, указав ему любой стандартный тип поля, но обязательно передать параметры *read\_only=Tru*e и *default=<значение\_по\_умолчанию>* (для поля **owner** это будет *default=serializers.CurrentUserDefault()*).

class CatSerializer(serializers.ModelSerializer):

    ...

    owner = serializers.PrimaryKeyRelatedField(

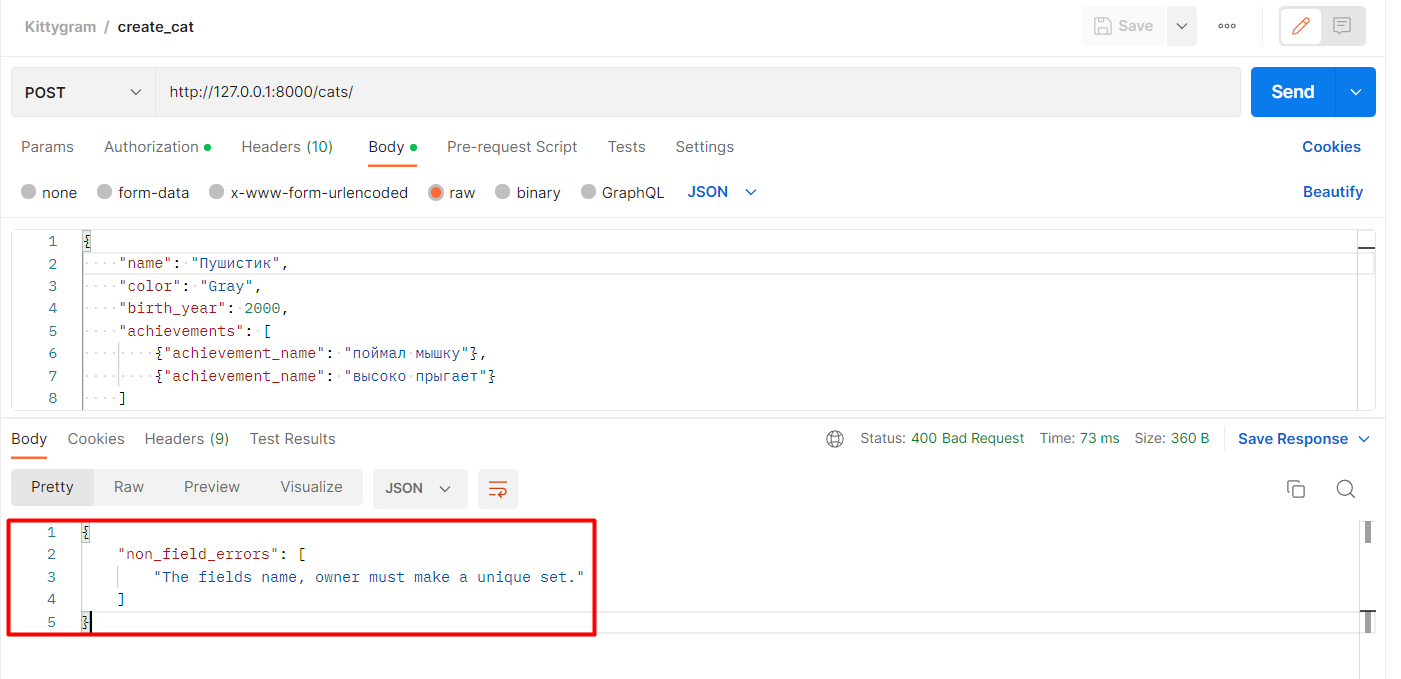
        read\_only=True, default=serializers.CurrentUserDefault())

    ...

В этом варианте поле owner может быть использовано на выходе сериализатора, но без передачи его значения в метод **save()** во вьюсете тут уже не обойтись.

В обоих случаях проблема будет решена, новые записи начнут добавляться в БД после обработки запросов и валидация **UniqueTogetherValidator** будет работать корректно.

Попробуем дважды добавить одного и того же котика; при попытке добавить его второй раз получим сообщение об ошибке:



**Важно**: класс **UniqueTogetherValidator** всегда накладывает неявное ограничение: все поля сериализатора, к которым применён этот валидатор, обрабатываются как обязательные. Поля со значением **default** — исключение: они всегда предоставляют значение, даже если пользователь не передал их в запросе.

# 16 Проверка прав: Permissions

Определение прав доступа — важная задача при разработке сервиса; если не озаботиться этим вопросом заранее, то возникнет риск, что любой пользователь сможет получить полный доступ ко всем функциям вашего сервиса.

В последней версии API проекта Kittygram любой аутентифицированный пользователь может удалять и редактировать не только свои, но и чужие записи. Это противоречит как обычной логике, так и техзаданию.

Мало того. Хозяева котиков очень хотят, чтобы об их любимцах узнало как можно больше людей, а доступ на чтение сейчас есть только у аутентифицированных пользователей; это сужает аудиторию. Значит, надо дать доступ на чтение к эндпоинту */cats/* всем, даже анонимам. Остальные эндпоинты должны быть доступны только зарегистрированным пользователям.

**Разрешения**

Для управления правами доступа в DRF есть встроенные классы, они называются **permissions** (по-русски их называют «*пермишены*», от англ. *permissions*, «*разрешения*»). Права доступа проверяются при получении запроса, одновременно с аутентификацией: система определяет, достаточно ли у пользователя прав на выполнение запрошенных операций.

Эта проверка выполняется в самом начале обработки запроса.

Настроить права доступа можно **на уровне** **всего проекта** или **на уровне отдельных классов и функций**. Распространённый подход состоит в комбинировании этих настроек: устанавливаются глобальные ограничения на уровне проекта, а при необходимости ограничения ослабляются на уровне классов или функций.

**Разрешения на уровне проекта**

Чтобы определить разрешения на уровне проекта, в словаре настроек **REST\_FRAMEWORK** задают параметр **DEFAULT\_PERMISSION\_CLASSES**. Вы уже делали это, когда подключали в Kittygram аутентификацию по токену:

REST\_FRAMEWORK = {

    ...

    'DEFAULT\_PERMISSION\_CLASSES': [

        'rest\_framework.permissions.IsAuthenticated',

    ],

}

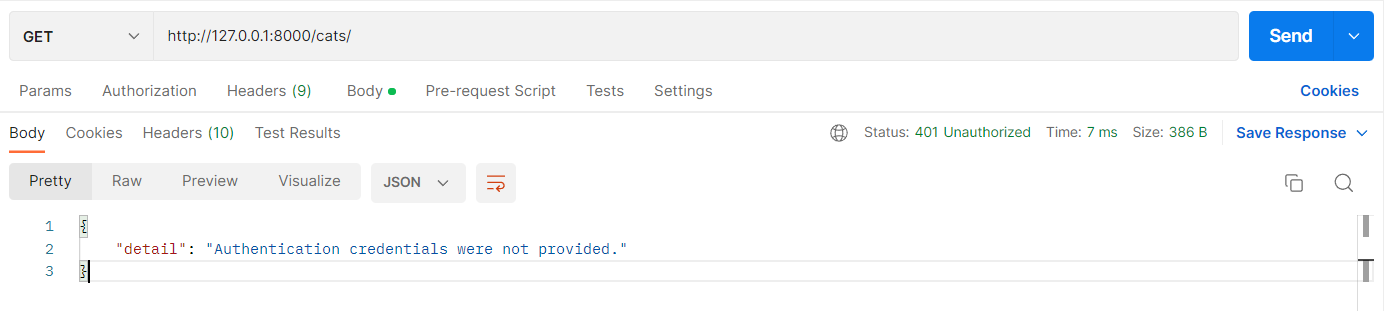
На уровне проекта можно установить один из четырёх вариантов доступа:

* **AllowAny** — всё разрешено, любой пользователь (и даже аноним) может выполнить любой запрос.
* **IsAuthenticated** — только аутентифицированные пользователи имеют доступ к API и могут выполнить любой запрос. Остальным вернётся ответ "*401 Unauthorized*".
* **IsAuthenticatedOrReadOnly** — то же что и в предыдущем доступе, но анонимы могут делать запросы на чтение; запросы на создание, удаление или редактирование информации доступны только аутентифицированным пользователям.
* **IsAdminUser** — выполнение запросов запрещено всем, кроме пользователей с правами администратора, тех, для которых свойство **user.is\_staff** равно *True*.

Важная особенность ограничения прав на уровне проекта состоит в том, что если ограничение в **settings.py** не прописано явным образом, то по умолчанию устанавливается доступ **AllowAny**.

Почитайте на досуге [официальную документацию](https://www.django-rest-framework.org/api-guide/permissions/#setting-the-permission-policy) — там много интересных примеров.

После подключения аутентификации по токену в Kittygram установлен вариант доступа **IsAuthenticated**. При таком уровне ограничений попытка анонимно запросить, например, даже список котиков, закончится неудачей.



Однако часть нашей задачи — предоставить доступ на чтение анонимным пользователям. А вот изменять объекты анонимы не должны, такие запросы надо разрешить только аутентифицированным пользователям — авторам объектов.

Для начала следует изменить настройки **settings.py**: установить уровень доступа **IsAuthenticatedOrReadOnly**:

...

REST\_FRAMEWORK = {

    ...

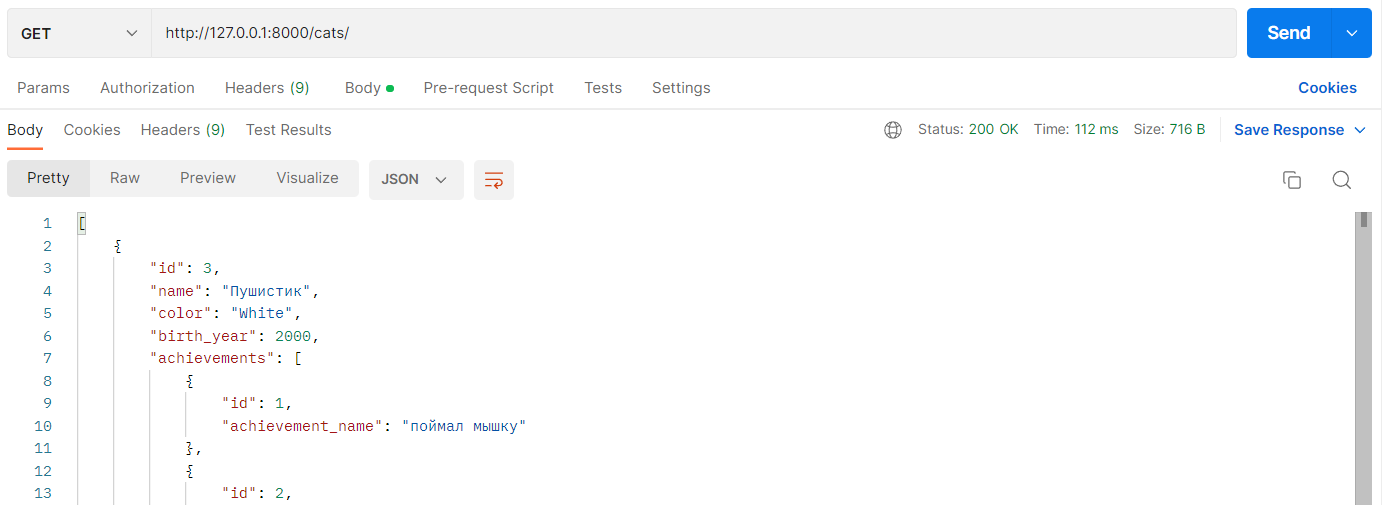
    'DEFAULT\_PERMISSION\_CLASSES': [

        'rest\_framework.permissions.IsAuthenticatedOrReadOnly',

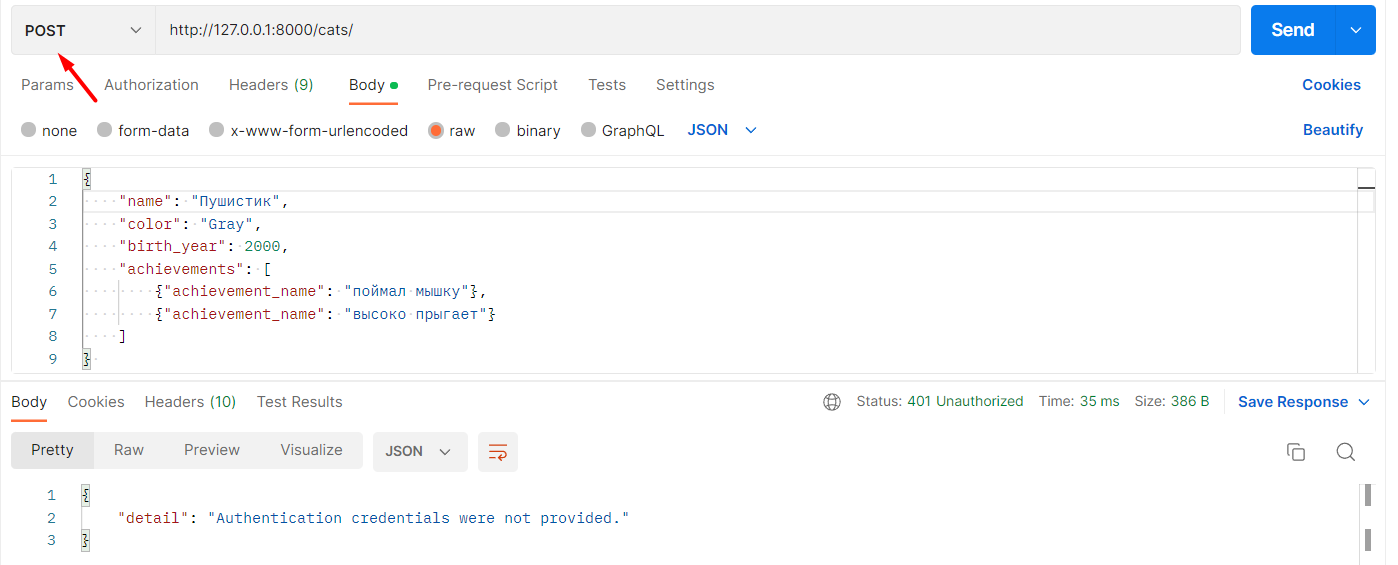
    ],

}

Теперь анонимный GET-запрос на получение коллекции котиков будет обработан успешно:



А запрос на изменение информации без токена не сработает.



**Разрешения на уровне представления**

Анонимный доступ на чтение теперь предоставлен ко всем эндпоинтам, но задача была в ином: анонимный доступ на чтение — только к котикам, а к другим данным, например, к перечню зарегистрированных хозяев — по токену.

Это ограничение лучше организовать так:

* на уровне проекта установить доступ ко всему API только по токену:

REST\_FRAMEWORK = {

    ...

    'DEFAULT\_PERMISSION\_CLASSES': [

        'rest\_framework.permissions.IsAuthenticated',

    ],

}

* на уровне вьюсета, к которому нужен анонимный доступ, установить разрешение **IsAuthenticatedOrReadOnly**. Для этого нужно импортировать модуль **permissions** из пакета **rest\_framework** и добавить атрибут **permission\_classes** в тело класса. Дописываем **CatViewSet**:

from rest\_framework import permissions

...

class CatViewSet(viewsets.ModelViewSet):

    queryset = Cat.objects.all()

    serializer\_class = CatSerializer

    # Устанавливаем разрешение

    permission\_classes = (permissions.IsAuthenticatedOrReadOnly,)

Теперь анонимным GET-запросом по-прежнему можно получить информацию о котиках. Но информация о пользователях Kittygram для анонимов будет под замком.

**Приоритеты разрешений**

У разрешений на уровне проекта приоритет ниже, чем у разрешений на уровне представления. Именно это позволяет локально переопределять права доступа там, где это необходимо.

У разрешений, установленных на одном уровне, тоже есть приоритеты: у более строгих приоритет выше. Если во вьюсете в атрибуте **permission\_classes** указать разрешения **IsAuthenticatedOrReadOnly** и **IsAdminUser**, то всем, кроме пользователей с правами **админа**, будут запрещены любые запросы: у более строгого разрешения — приоритет.

**Создание собственных разрешений**

Промежуточный итог: сделать запрос к котикам теперь может любой желающий, запросы о владельцах котов и запросы на изменение или удаление объектов доступны только аутентифицированным пользователям, — отлично, всё по плану.

Но вот незадача: изменить или удалить информацию может любой владелец действующего токена, а не только автор, создавший объект.

Задача в том, чтобы дать разрешение на редактирование или удаление объектов модели Cat только тому пользователю, который создал этот объект; всем прочим пользователям и анонимам информация должна быть доступна только для чтения.

Встроенный пермишен **IsAuthenticatedOrReadOnly** не справится с такой задачей, придётся удалить его из **CatViewSet**. Напишем своё разрешение, в DRF есть такая возможность.

**Методы базового класса BasePermission**

Разрешения в DRF можно разделить на два типа:

* разрешения «на уровне запроса» определяют, разрешён ли в общем сделанный запрос: разрешён ли определённый тип запроса, есть ли разрешение для пользователя или для его IP, разрешено ли делать запросы в это время года — могут быть любые условия, касающиеся самого запроса, а не запрашиваемой информации;
* разрешения «на уровне объекта» касаются запрошенных данных: имеет ли право пользователь сделать запрос к определённым данным, является ли пользователь автором этих данных, можно ли к запрошенным данным отправлять запросы определённого типа — для этих разрешений тоже можно установить широкий спектр условий.

В Django REST Framework все классы разрешений наследуются от базового класса **BasePermission**. В нем описаны два метода:

* в методе **has\_permission** определяются разрешения на уровне запроса;
* в методе **has\_object\_permission** устанавливаются разрешения на уровне объекта.

Доступ будет разрешён, если методы вернут *True*.

class BasePermission(metaclass=BasePermissionMetaclass):

    # Определяет права на уровне запроса и пользователя

    def has\_permission(self, request, view):

        return True

    # Определяет права на уровне объекта

    def has\_object\_permission(self, request, view, obj):

        return True

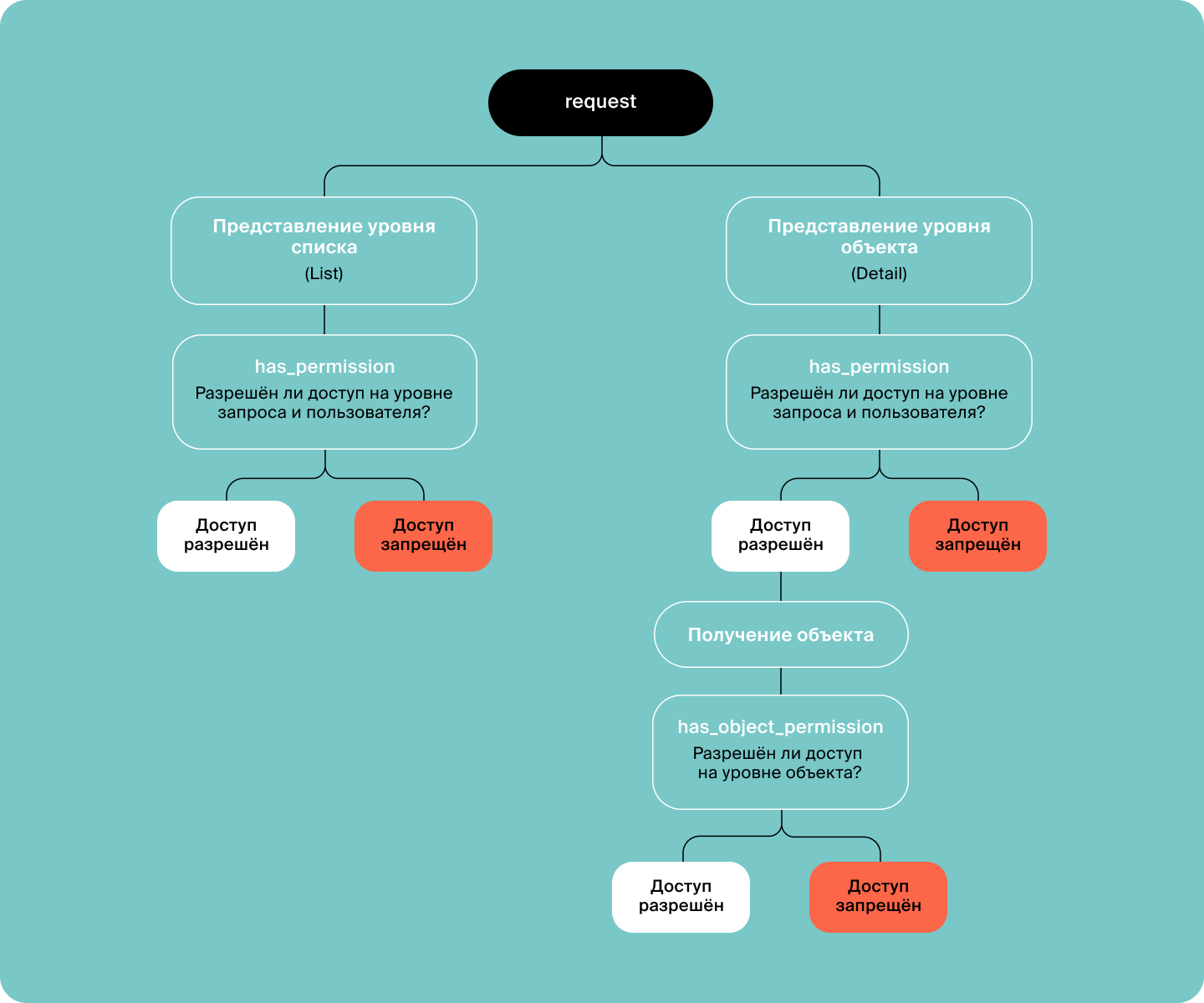
И тот, и другой метод обладают информацией о запросе. А **has\_object\_permission** обладает данными и об объекте запроса.

Чтобы создать собственное разрешение — нужно описать свой класс, расширяющий **BasePermission** и переопределить один или оба его метода.

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.

Если же переопределить только один из этих методов — окончательное решение о доступе будет зависеть от результата работы того метода, который был переопределён. В общем виде алгоритм определения разрешений после получения запроса выглядит так:



Теорию преодолели, теперь решим поставленную задачу — напишем кастомный пермишен **OwnerOrReadOnly**, который расширит возможности встроенных пермишенов и разрешит полный доступ к объекту только автору. Разрешения для анонимов пока настраивать не будем: займёмся ими после того, как разберёмся с аутентифицированными пользователями.

В структуре проекта собственные пермишены принято описывать в отдельном файле *permissions.py*.

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

Этот пермишен работает так:

* Сначала в **has\_permission()** проверяется метод запроса и статус пользователя. Если метод запроса безопасный (то есть GET, HEAD или OPTIONS) или если пользователь аутентифицирован (то есть предоставил валидный токен), то метод вернет *True*. В этом методе доступа к объекту запроса нет, поэтому мы не знаем и никак не можем проверить, является ли пользователь, делающий запрос, автором объекта.
* Если **has\_permission()** вернул *True*, то после получения объекта вызывается метод **has\_object\_permission()**, в него передаётся запрошенный объект, и теперь в этом методе можно проверить, совпадает ли автор объекта с пользователем из запроса.

Такой пермишен имеет смысл применять только к тем вьюсетам, модели которых содержат поле **owner**; если этого поля нет — пермишен будет бесполезен.

Вьюсет **CatViewSet** работает с моделью **Cat**, значит, при проверке в методе **has\_object\_permission()** в объект **obj** будет передаваться запрошенный экземпляр модели **Cat**. В этой модели есть поле **owner**, и при проверке будет проведено сравнение пользователя из запроса и содержимого этого поля.

Теперь пермишен **OwnerOrReadOnly** можно подключить к вьюсету **CatViewSet**:

...

from .permissions import OwnerOrReadOnly

class CatViewSet(viewsets.ModelViewSet):

    queryset = Cat.objects.all()

    serializer\_class = CatSerializer

    permission\_classes = (OwnerOrReadOnly,)

Итоговое разрешение на обработку конкретного запроса будет зависеть от того, что за запрос получен:

* запрос для получение списка объектов или создания нового объекта
* или запрос для получения, изменения или удаления конкретного объекта

Задача выполнена, полный доступ к объектам модели **Cat** предоставлен только хозяевам котиков.

Выполните через Postman запросы к различным котикам с токенами разных пользователей. Попробуйте перекрасить котика, отправив запрос с токеном пользователя, который не является его хозяином. Если всё сделано правильно, то такой запрос не будет обработан.

**Различные пермишены для одного вьюсета**

Для одного и того же вьюсета можно применять различные пермишены, например — в зависимости от типа запроса.

Пермишен **OwnerOrReadOnly** настроен так, что при работе именно с вьюсетом при анонимном запросе позволяет получать список котиков, но не позволит получить информацию о конкретном котике. Исправим это. Опишем еще один кастомный пермишен в **permissions.py**; назовём его **ReadOnly**:

class ReadOnly(permissions.BasePermission):

    def has\_permission(self, request, view):

        return request.method in permissions.SAFE\_METHODS

Во вьюсете **CatViewSet** определим метод **get\_permissions**: зададим условия, при которых должен применяться тот или иной пермишен:

class CatViewSet(viewsets.ModelViewSet):

    queryset = Cat.objects.all()

    serializer\_class = CatSerializer

    def get\_permissions(self):

    # Если в GET-запросе требуется получить информацию об объекте

        if self.action == 'retrieve':

        # Вернем обновленный перечень используемых пермишенов

            return (ReadOnly(),)

        # Для остальных ситуаций оставим текущий перечень пермишенов без изменений

        return super().get\_permissions()

Теперь при GET-запросе информации о конкретном котике доступ будет определятся пермишеном **ReadOnly**: запросы будут разрешены всем. При остальных запросах доступ будет определять пермишен **OwnerOrReadOnly**.

# 17 Пагинация в API

Рост популярности Kittygram выявил и другие интересные моменты. Сейчас в базе хранится значительное количество информации о разных котиках. В результате GET-запрос к эндпоинту */cats/* вернёт сразу 100500 объектов.

Пользователи недовольны, у них проблемы: API возвращает гору информации, в которой сложно разобраться; пользователям обычно не нужен весь список котиков, но ограничить количество объектов в ответе невозможно; обработка большого ответа требует дополнительного времени и ресурсов.

**Пагинация на уровне проекта**

В составе DRF по умолчанию доступны несколько классов пагинации. Начнём с самого простого класса: **PageNumberPagination**. Он делит выдачу на части и, дополнительно, добавляет в ответ API счётчик страниц и ссылки на предыдущую и следующую «страницы» с результатами.

Пагинацию можно включить на уровне всего проекта, добавив 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 = {

    ...

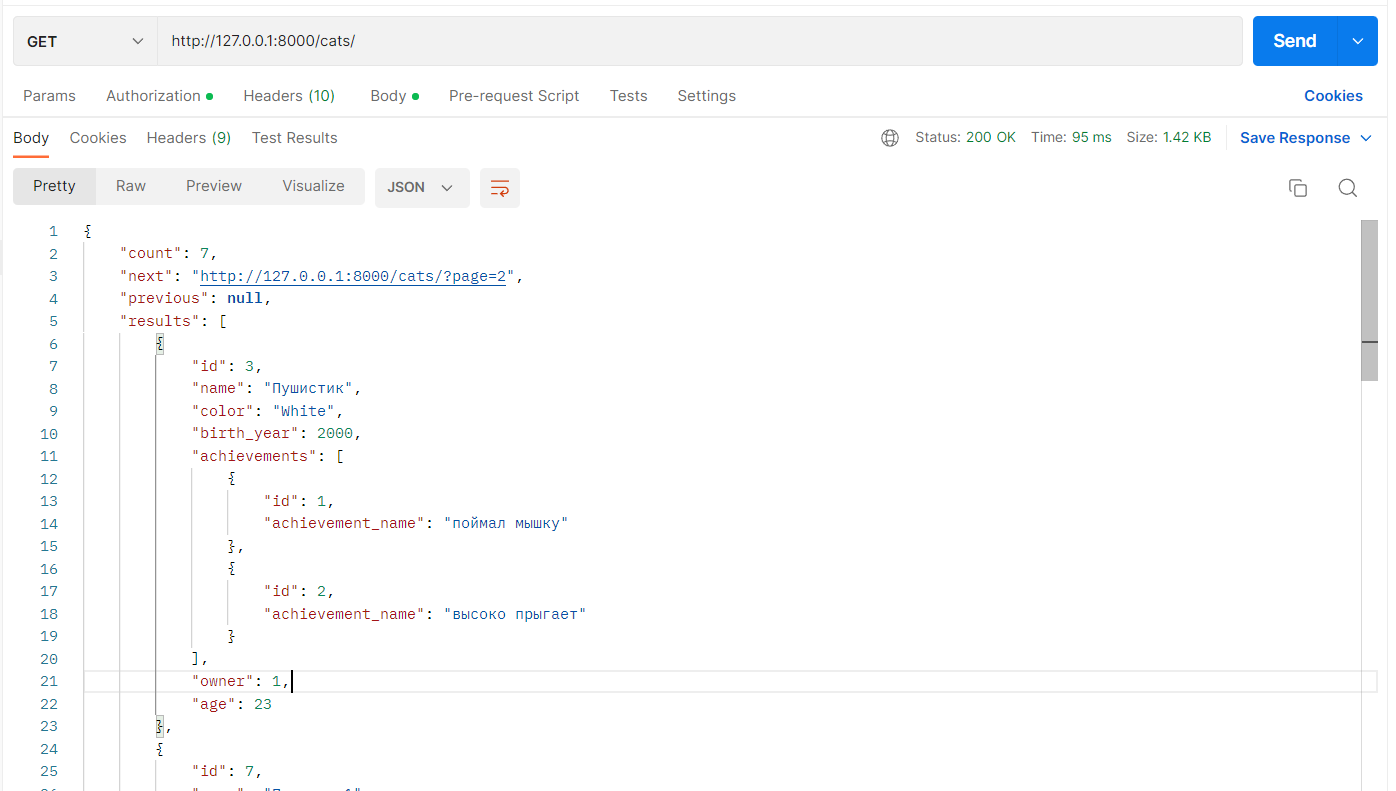
    'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.PageNumberPagination',

    'PAGE\_SIZE': 5,

}

Пагинация «*из коробки*» будет работать только для **дженериков** и **вьюсетов**. Для view-классов пагинацию настраивают иначе; детали реализации можно [подсмотреть](https://github.com/encode/django-rest-framework/blob/98e56e0327596db352b35fa3b3dc8355dc9bd030/rest_framework/generics.py#L24) в исходном коде классов **mixins.ListModelMixin** и **generics.GenericAPIView**.

В настройках пагинации мы установили выдачу пяти объектов на «страницу». Если в БД есть более пяти записей о котиках, то при GET-запросе к эндпоинту */cats/* вернётся примерно такой ответ:



Обратите внимание: если раньше список объектов был прямо в теле JSON, то теперь объекты вложены в список **results**.

**Пагинация на уровне view-класса**

Пагинацию можно установить для отдельного **view-класса** (для **Generics** или **Viewsets**), указав класс пагинатора в атрибуте **pagination\_class**. Если пагинация одновременно установлена и на уровне проекта, и на уровне класса — приоритет будет у настроек класса.

from rest\_framework.pagination import PageNumberPagination

class CatViewSet(viewsets.ModelViewSet):

    queryset = Cat.objects.all()

    serializer\_class = CatSerializer

    permission\_classes = (permissions.IsAuthenticatedOrReadOnly,)

    pagination\_class = PageNumberPagination

Для **PageNumberPagination**, установленного в классе, параметр **PAGE\_SIZE** будет взят из словаря **REST\_FRAMEWORK** в **settings.py**.

Если пагинация установлена на уровне проекта, то для отдельного класса её можно отключить, установив для атрибута **pagination\_class** значение **None**.

**Новая структура ответа**

Добавление пагинации изменило структуру выдачи, и если пользователи Kittygram или другие сервисы уже пользуются нашим API — у них будут проблемы: после таких изменений извлечь данные из выдачи не удастся, придётся переписывать обработчики.

Другое важное изменение состоит в том, что при включённой пагинации запрос к API теперь можно делать с дополнительным параметром **page**:

*GET* [*http://127.0.0.1:8000/cats/?page=2*](http://127.0.0.1:8000/cats/?page=2)

Значением этого параметра должно быть целое число, указывающее на нужную «*страницу*» выдачи. Нумерация «*страниц*» начинается с **единицы**.

У объекта выдачи также появились поля **count**, **next** и **previous**: это общее количество объектов и URL'ы следующей и предыдущей страниц.

**Класс LimitOffsetPagination**

Помимо **PageNumberPagination** есть и более гибкий класс для пагинации: **LimitOffsetPagination**.

При работе с классом PageNumberPagination разработчик жёстко устанавливает разбиение по страницам, а класс **LimitOffsetPagination** даёт возможность клиенту самостоятельно определять, какое число объектов вернётся (параметр **limit**) и с какого по счёту объекта начать отсчёт (параметр **offset**).

Подключим этот класс к **CatViewSet** и посмотрим на результат его работы:

...

from rest\_framework.pagination import LimitOffsetPagination

class CatViewSet(viewsets.ModelViewSet):

    queryset = Cat.objects.all()

    serializer\_class = CatSerializer

    permission\_classes = (OwnerOrReadOnly,)

    # Даже если на уровне проекта установлен PageNumberPagination

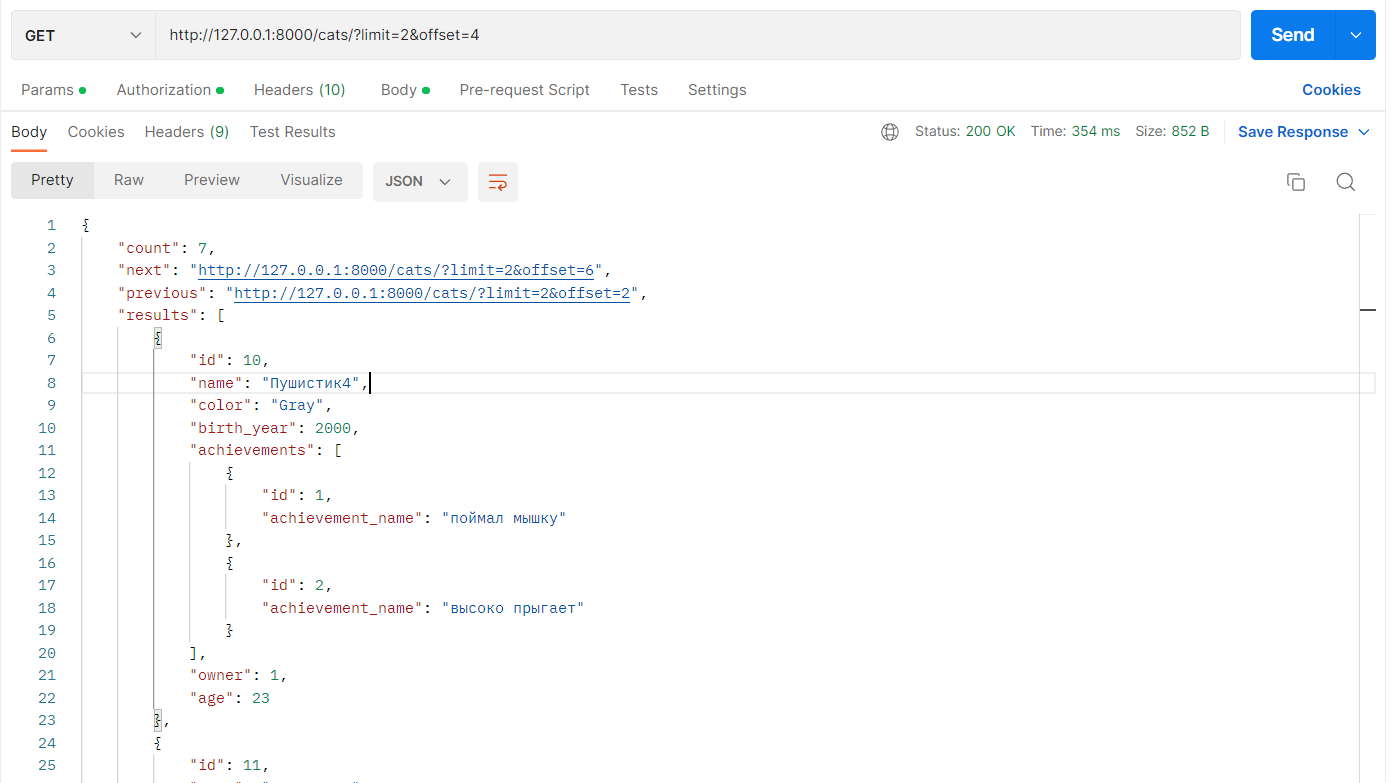
    # Для котиков будет работать LimitOffsetPagination

    pagination\_class = LimitOffsetPagination

При пагинации посредством **LimitOffsetPagination** GET-запрос должен выглядеть примерно так:

*GET* [*http://127.0.0.1:8000/cats/?limit=2&offset=4*](http://127.0.0.1:8000/cats/?limit=2&offset=4)

Такой GET-запрос вернёт два объекта, с пятого по шестой (или меньше, если в результате запроса менее 6 объектов).



Для решения большинства стандартных задач вполне хватает классов **PageNumberPagination** и **LimitOffsetPagination**, но в DRF есть и другие пагинаторы; узнать о них можно в [документации](https://www.django-rest-framework.org/api-guide/pagination/#api-reference).

**Пользовательский класс пагинации**

Класс **PageNumberPagination** настраивается в **settings.py**, и указать там отдельное значение для конкретного **вьюсета** невозможно. Даже если прописать пагинатор прямо во вьюсете — размер выдачи он всё равно возьмёт из настроек проекта.

Если все эндпоинты должны возвращать одинаковое количество объектов на странице — проблем нет. Но тимлид не был бы тимлидом, если бы не придумывал постоянно новые задачи:

— Нам нужно изменить выдачу: котиков возвращаем по двадцать на странице, а все остальные объекты — по пять.

Готовые классы для этой задачи не подходят, будем писать собственный.

Кастомный пагинатор наследуют от подходящего встроенного класса или от базового **BasePagination**. После этого в кастомном пагинаторе можно переопределить необходимые параметры. Всё знакомо.

Приступим: опишем собственный класс **CatsPagination**, унаследованный от **PageNumberPagination**, и переопределим нужные параметры: нам нужен только **page\_size**.

# cats/pagination.py

from rest\_framework.pagination import PageNumberPagination

class CatsPagination(PageNumberPagination):

    page\_size = 20

Пагинатор **CatsPagination** готов, его можно указывать в настройках пагинации во вьюсетах, дженериках или в глобальных настройках проекта.

from .pagination import CatsPagination

class CatViewSet(viewsets.ModelViewSet):

    queryset = Cat.objects.all()

    serializer\_class = CatSerializer

    permission\_classes = (permissions.IsAuthenticatedOrReadOnly,)

    # Вот он наш собственный класс пагинации с page\_size=20

    pagination\_class = CatsPagination

**Методы paginate\_queryset и get\_paginated\_response**

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**.

При описании собственных классов-пагинаторов эти методы тоже можно переопределять — например, если нужно изменить структуру ответа или названия полей в нём. [Примеры из документации](https://www.django-rest-framework.org/api-guide/pagination/#example) вам помогут.