DJANGO. DJANGO REST FRAMEWORK

Оглавление

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.

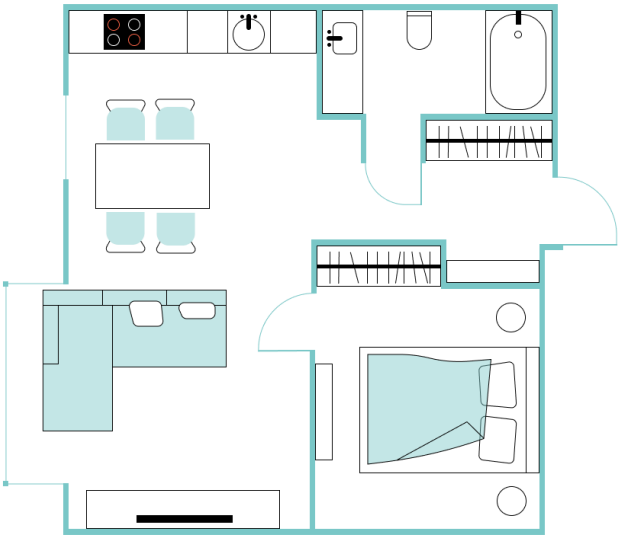
Можно сделать всё самостоятельно. Изготовить кирпичи и сложить стены. Напилить досок и сделать окна, двери и табуретки. Не забыть бы про плитку в ванной, трубы и тысячу других вещей.

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 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.

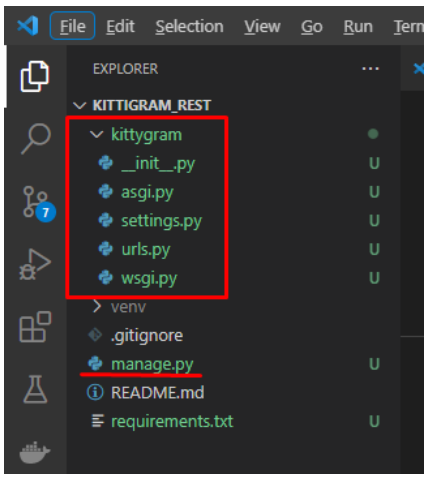
- **Работа с базами данных**
  С базами данных Django общается через инструмент Django ORM (объектно-реляционное отображение). Это «переводчик» с языка Python на язык SQL, понятный большинству баз данных. Благодаря Django ORM все запросы к базе данных можно писать прямо на Python.

2 Содержимое директории проекта

После выполнения команды создания базовой структуры проекта:

$ django-admin startproject kittygram .

вы получите следующую структура проекта:



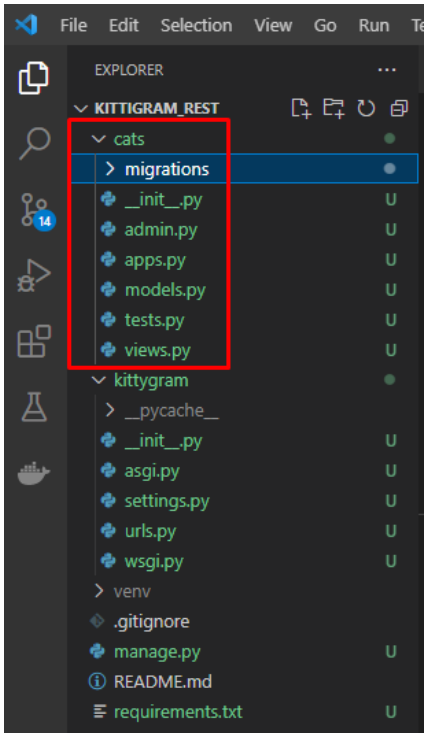
- В директории ***kittygram_rest/kittygram*** лежат файлы с кодом проекта.
- ***kittygram_rest/manage.py*** — файл управления Django-проектом из командной строки. Вы часто будете к нему обращаться.
- В файле ***kittygram_rest/kittygram/urls.py*** настраиваются URL проекта.
- ***kittygram_rest/kittygram/wsgi.py*** — 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.

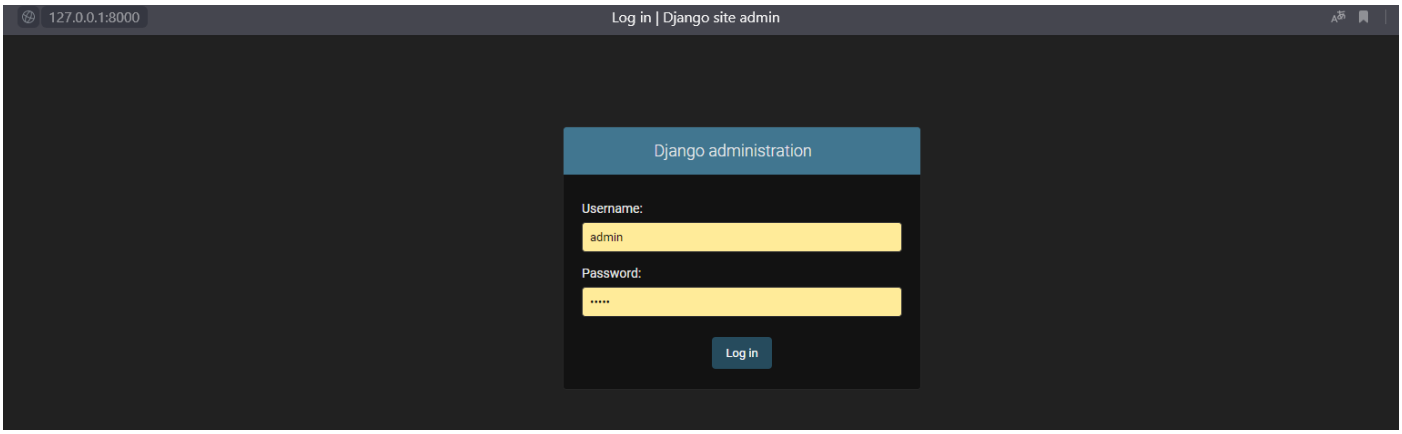
Именно эти приложения позволят нам создать учётную запись администратора сайта и авторизоваться на сайте. При создании администратора мы дадим ему максимум прав; такие аккаунты в Django называются «суперпользователями» (superuser).

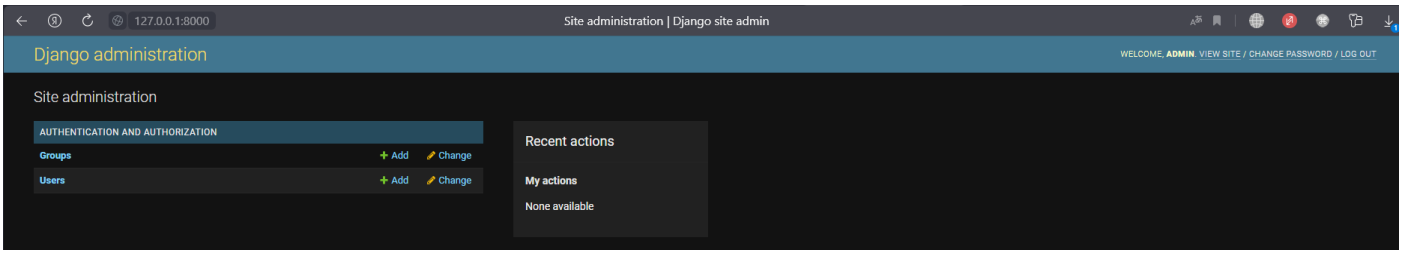
Под этой учётной записью вы будете управлять сайтом через интерфейс администрирования (админ-зону).

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

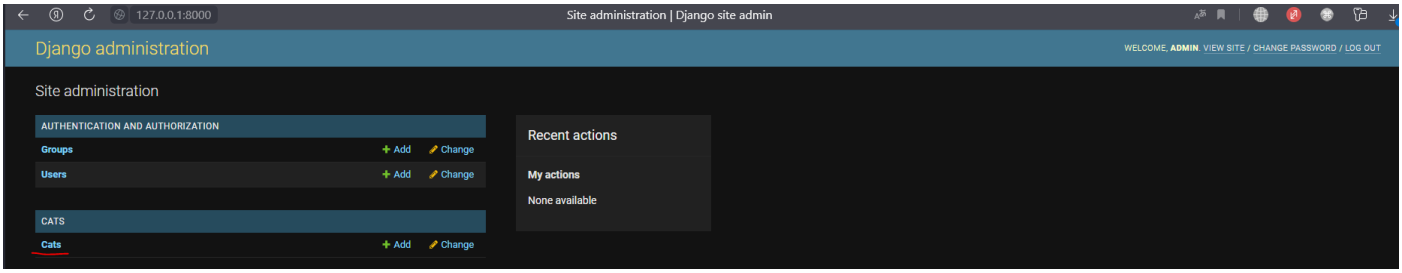
добавилось много таблиц. Но в интерфейсе админки видны только две модели: Groups и Users. Остальные модели — служебные, они не требуют внимания администратора и потому исключены из интерфейса.

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

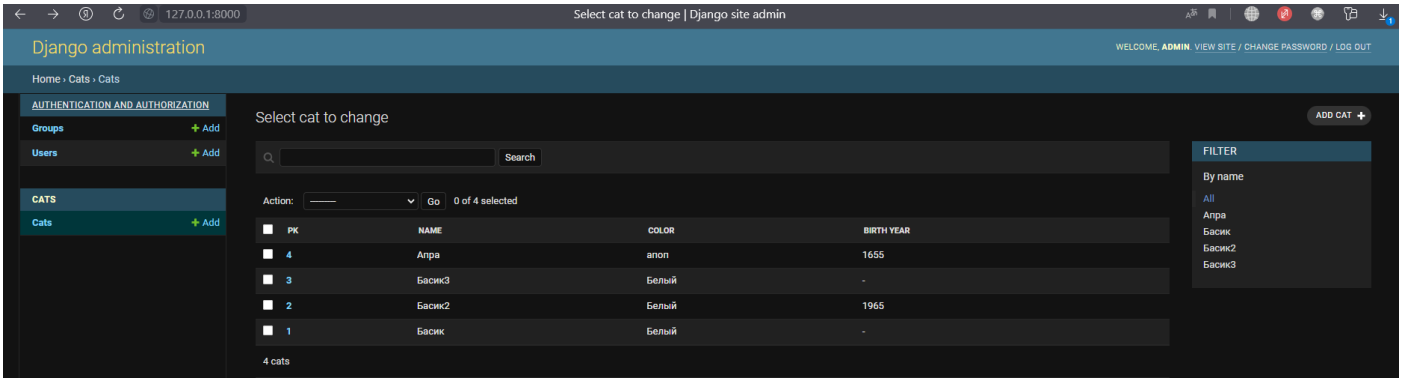
Для настройки отображения модели в интерфейсе админки применяют класс **ModelAdmin**. Он связывается с моделью и конфигурирует отображение данных этой модели. В этом классе можно настроить параметры отображения.

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:

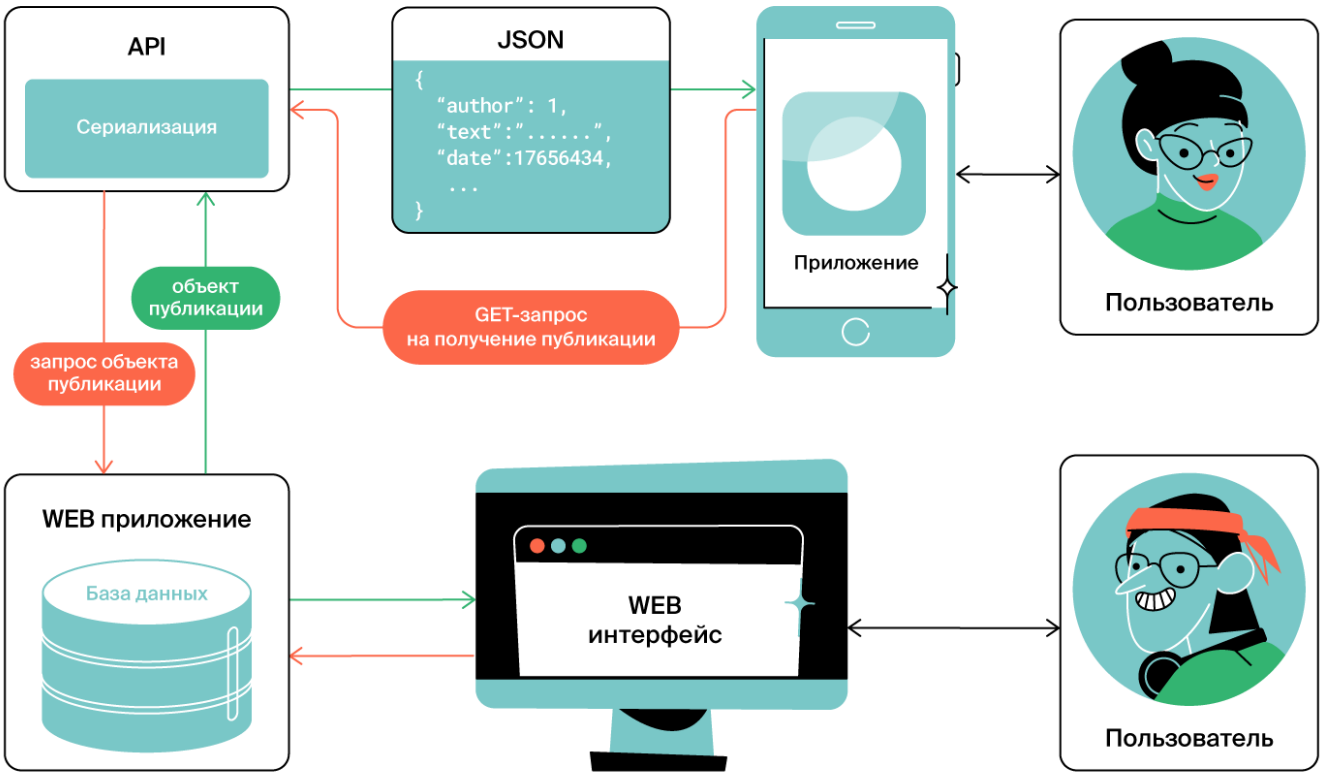
- Мобильное приложение отправит *GET-запрос* к **API**; в запросе будет передан **id** поста.
- API сделает запрос к базе данных проекта в надежде получить оттуда экземпляр класса **Post** с заданным **id**.
- Если пост с таким **id** есть в БД, то программа получит объект класса **Post** (этот объект 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

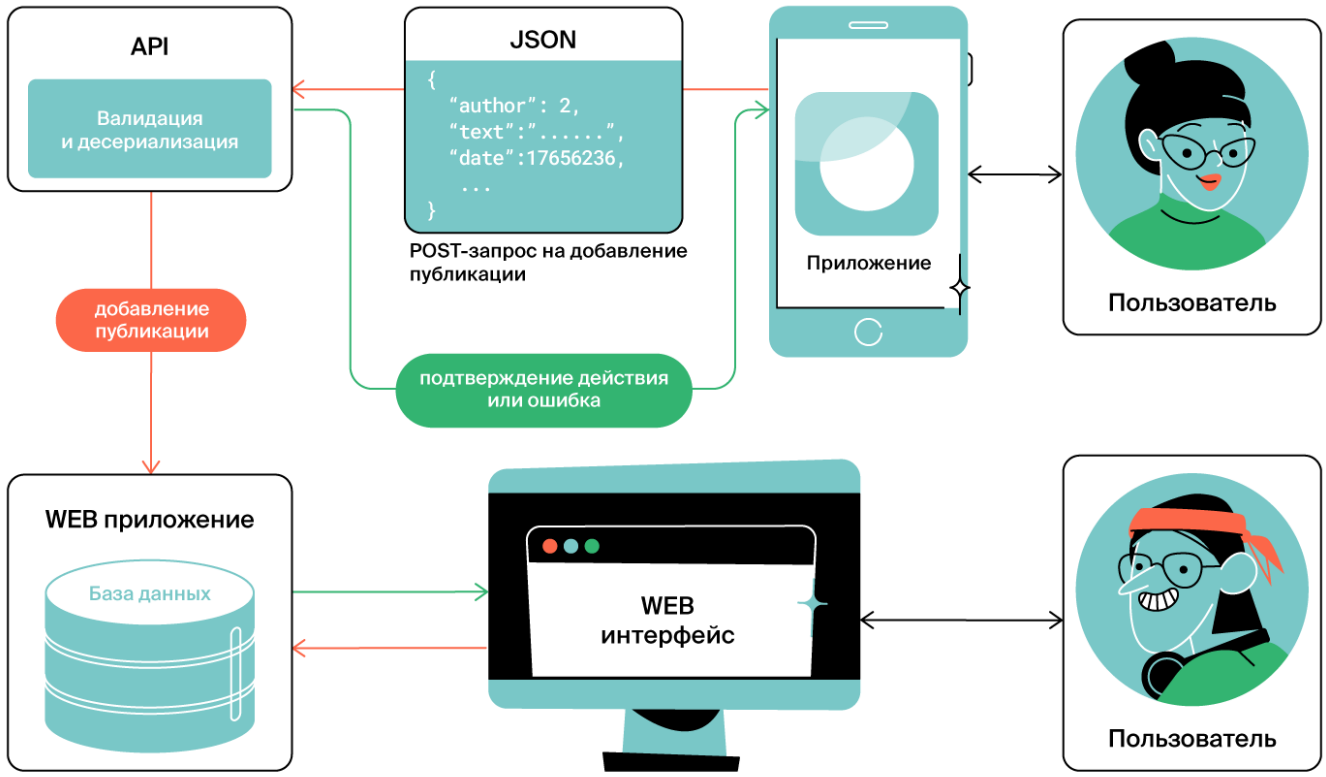
Если же полученные данные не пройдут валидацию, то API не будет добавлять их в базу данных, а просто отправит ответ с сообщением об ошибке.

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.

```python
# 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.

```python
# 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.

```python
# 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)

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:

```python
# 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().**

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

имена эндпоинтов будут такими:

```python
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, доступные 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.

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
```