

Django Development

Оглавление

Часть 1 - Основы Django

01_Подготовка окружения Django проекта

02_Создание основы проекта

03_Создание первого приложения

04_Django ORM

05_Работа с миграциями

06_Создание суперпользователя

07_Регистрация модели Post в админке

08_Загрузка тестовых данных

09_Проектирование структуры адресов

10_Добавление главной страницы

11_Шаблоны Django

12_Теги шаблонов Django - наследование и переопределение

13_Теги шаблонов Django - ветвления, циклы и ссылки

14_Шаблон главной страницы

15_Базовый шаблон

Часть 2 - Django продолжение

16_CRUD и фильтрация через ORM

17_Дополнительные возможности ORM

18_Агрегирующие функции в Django ORM

19_Управление пользователями

20_Generic Views

21_Добавление формы регистрации

22_Доработка базового шаблона

23_Язык сайта

24_Доработка шаблона формы

25_Создание фильтра

26_Шаблоны страниц входа и восстановления доступа

27_Эмуляция почтового сервера

28_Python - Декораторы

29_Декоратор login_required

30_Валидация форм

Часть 3 - Тестирование

31_Что такое тестирование

32_Запуск первых тестов

33_Подготовка условий для запуска тестов

34_Дополнительные компоненты Django

35_Паджинатор

36_Вспомогательные страницы flatpages

37_Профайл пользователя и страница записи

Часть 4 - Доработка проекта

38_Страницы с ошибками

39_Добавление картинок к постам

40_Комментарии

41_Рефакторинг шаблонов

42_Полезные функции

43_Оптимизация и кеширование

44_Бонус: установка django-debug-toolbar

Часть 5 - API

45_Что такое API. Формат JSON

46_API First. Архитектура REST

47_Правила именования ресурсов

48_Механизмы авторизации, протокол OAuth 2.0

49_Получение токена ВКонтакте

50_Как хранить секреты

51_Запрос к API сервиса ВКонтакте

52_Отправка SMS-уведомлений

53_Django Rest Framework

54_Сериализаторы

55_View-функции API

56_View-классы API

57_CRUD для Poemnotes

Часть 6 - Фильтрация и безопасность в API

58_Авторизация (проверка прав)

59_Throttling (ограничение количества запросов)

60_Пагинация

61_Фильтрация, сортировка и поиск

01_Подготовка окружения Django проекта

Многие задачи, с которыми вы столкнётесь, уже решены программистами, и их наработки доступны всем. Любой может включить в свой проект готовые модули, библиотеки или фреймворки. Эти наборы упаковывают в стандартизованные файлы, скачивание и установка таких файлов также стандартизированы и выполняются автоматически: программы-установщики знают, как с такими файлами обращаться. Такие стандартизованные «упаковки» называются **package** («пакет»), а программы-установщики — **пакетные менеджеры**.

В чём разница между библиотекой, модулем, фреймворком и пакетом?

- **Модуль** — это файл с кодом, подключаемый к нашему проекту. Обычно модуль отвечает за решение строго ограниченной задачи. Модули, которые занимаются какой-то одной темой, можно объединить в **библиотеку**. Например, библиотека для обработки картинок *Pillow* может работать со множеством форматов файлов. Исходный код *Pillow* разбит на множество модулей.
- **Фреймворк** — это более сложный вид библиотеки, набор готовых инструментов для решения распространённых задач. Если сравнивать программирование с производством, то без фреймворка вы сперва собираете станки, а затем на них начинаете выпуск продукции. А фреймворк поставляет вам готовые станки: подключи и работай. Но зачастую это чуть-чуть не те станки, какие бы вам хотелось.
- **Пакет** — это модуль, библиотека или даже фреймворк, упакованный по определённым правилам и подготовленный для того, чтобы программисты могли без проблем применить его в разработке.

Как только пакет опубликован, программисты всего мира принимаются использовать его. С этого момента начинается большая жизнь нового продукта.

Со временем в пакете обнаруживаются ошибки, у разработчиков возникают пожелания или предложения что-то исправить и расширить. В таких случаях авторы выпускают обновления пакетов, новые **версии**. Каждая новая версия обозначается уникальной строкой. Она обычно имеет вид «X.Y.Z», где X — главная (мажорная) версия, Y — номер обновления, Z — номер исправления. Но конкретные правила именования версий определяет автор пакета.

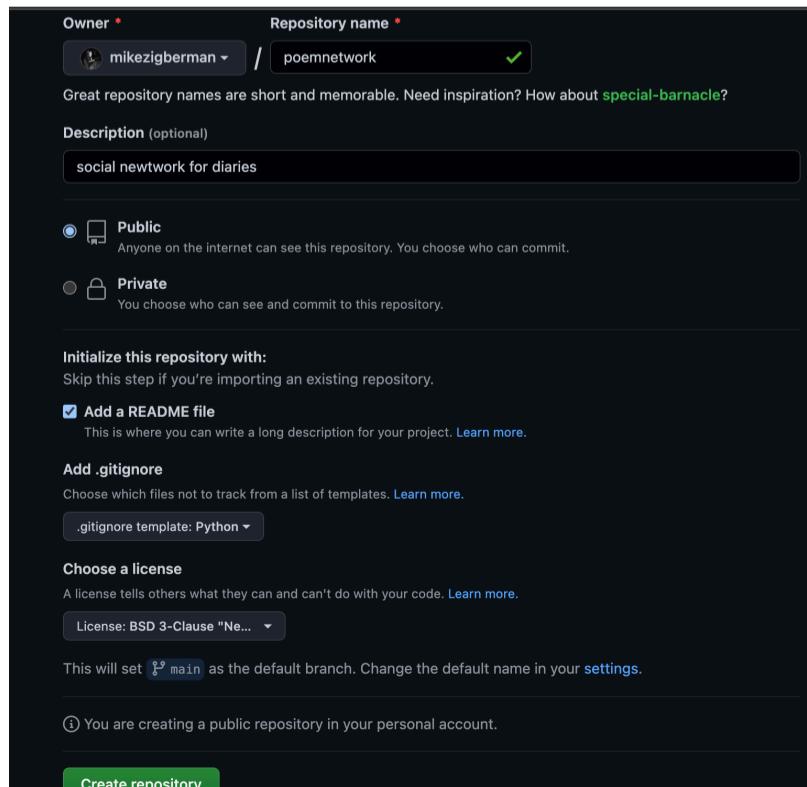
Подготовка git репозитория

Проект, над которым вы будете работать, станет частью вашего портфолио и будет доступен на сайте github.com. Если у вас ещё нет учетной записи на этом сайте — создайте её.

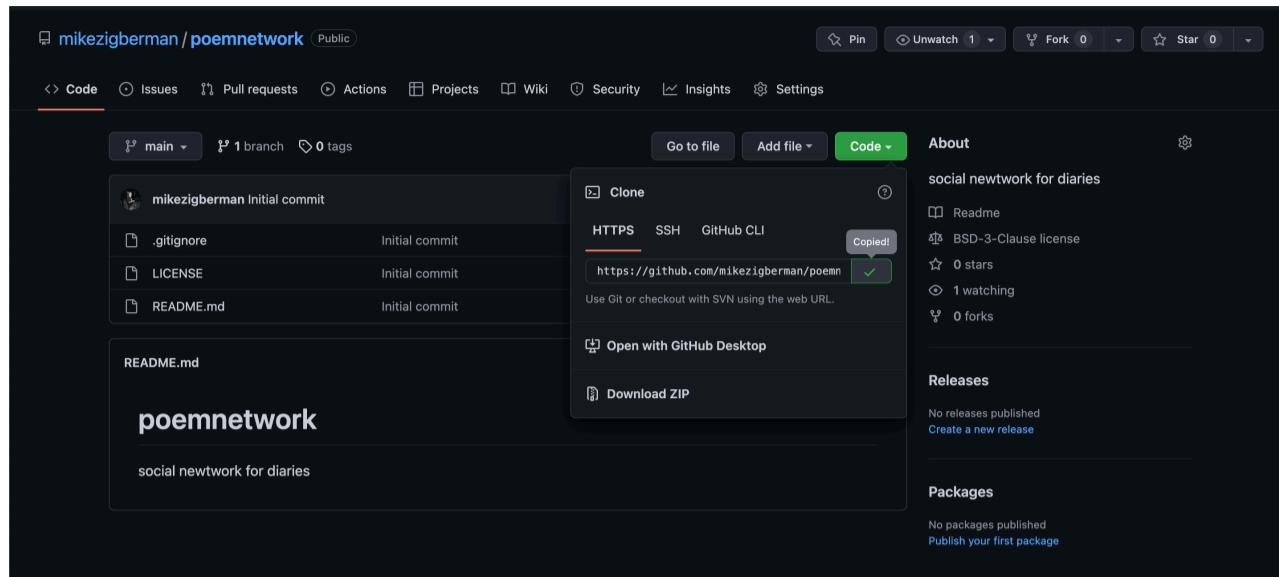
Обратите внимание: учебный проект может стать заметной частью вашего профессионального портфолио. Придумайте своей учетной записи имя, которое будет хорошо смотреться в резюме специалиста.

Создайте новый репозиторий на [сайте GitHub](https://github.com). Заполните следующие поля:

- Repository name — имя репозитория, назовите его **poemnetwork**.
- Description — описание проекта. Мы будем разрабатывать платформу для блогов, так что можно написать, например «Социальная сеть блогеров».
- Public/Private — тип репозитория: публичный или приватный. На время прохождения курса выберите **приватный**. По завершении проекта вы сможете самостоятельно поменять тип на «публичный».
- Initialize this repository with a README — создать файл README в репозитории. Включите эту опцию: в файле README можно дать описание проекта.
- Add .gitignore — добавить файл `.gitignore`. Это обязательный пункт, выберите в списке пункт `Python`.
- Add license — добавить лицензию. В лицензии вы устанавливаете права на свой проект. Мы рекомендуем указать лицензию BSD 3 или MIT: они предоставляют хороший баланс прав и ответственности.



После создания надо клонировать репозиторий локально. Скопируйте его адрес:



Откройте окно терминала Git Bash и перейдите в рабочую директорию Dev — вы создали её в предыдущих уроках.

Проверьте, что указываете правильный адрес папки, в которой будут храниться ваши проекты.

Под Windows выполните в терминале такие команды:

```
# Перейдите в рабочую директорию Dev
# Возможно, на вашем компьютере адрес немного иной
$ cd D:\Dev\
# Клонируйте репозиторий с сайта Github
$ git clone git@github.com:username/poemnetwork.git
```

Под macOS или Linux:

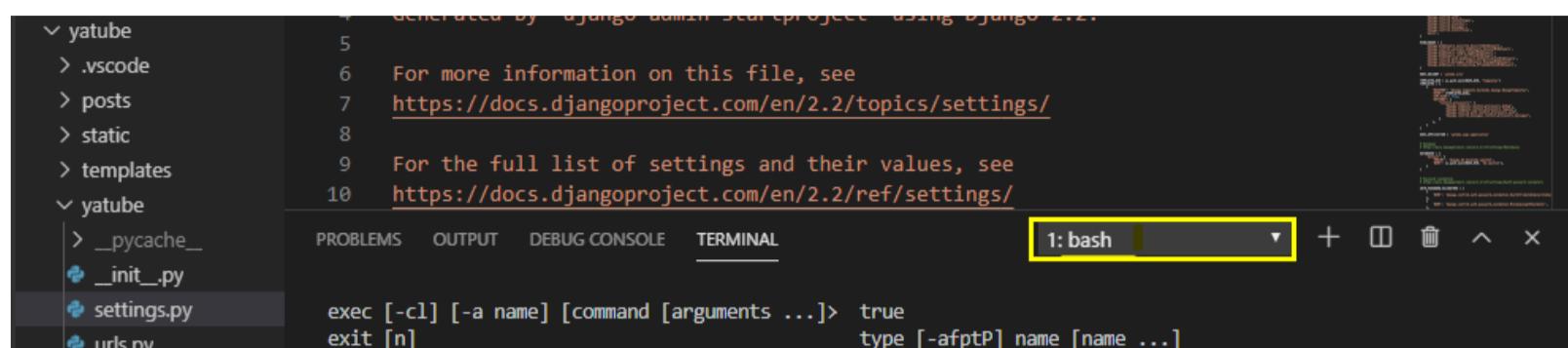
```
# Перейдите в рабочую директорию:
$ cd ~/Dev/
# Клонируйте репозиторий с сайта Github
$ git clone git@github.com:username/poemnetwork.git
```

После выполнения этих команд в папке Dev появится директория Yatube с файлами LICENSE и README.md. Это и будет **рабочая директория проекта**.

Встроенный терминал в Visual Studio Code

Окно терминала можно запускать прямо в интерфейсе Visual Studio Code. Если у вас на компьютере установлено несколько терминалов — в VSC можно выбрать, в каком именно терминале вы хотите работать. Это особенно актуально для Windows.

Запустите терминал (меню «Терминал» / «Новый терминал»). Проверьте, какой терминал запущен. Нам нужен **bash**.



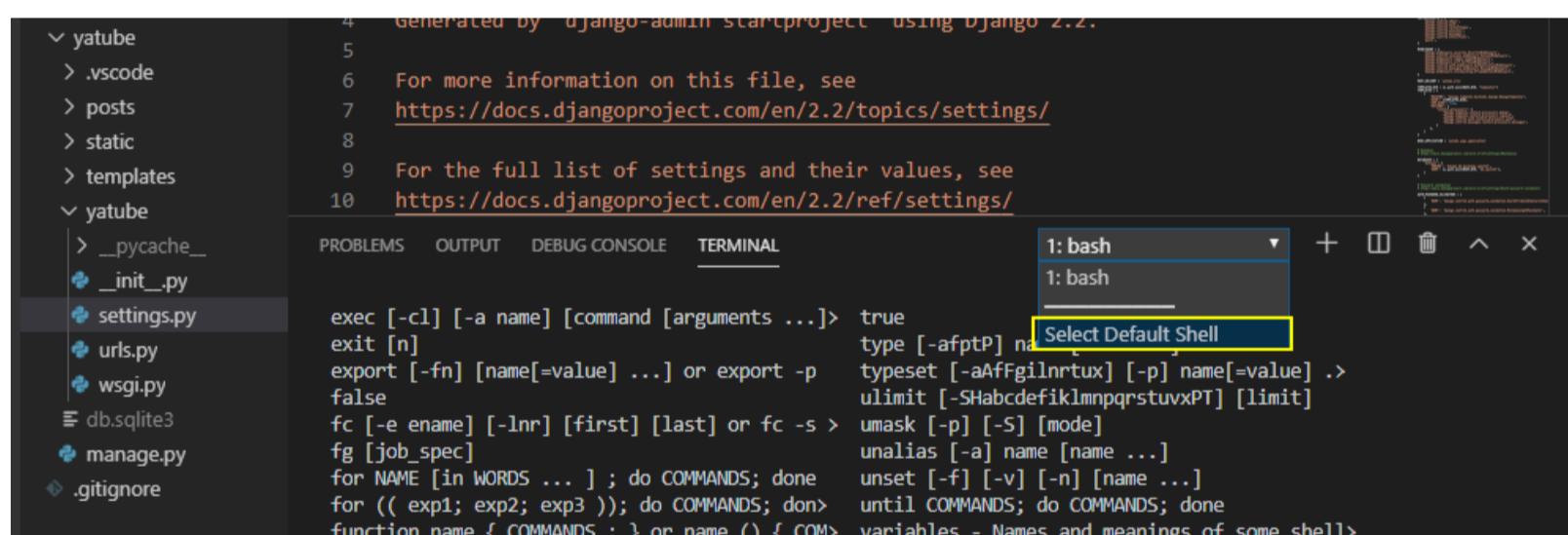
A screenshot of the VS Code interface. On the left is the Explorer sidebar showing a project structure with files like `settings.py` and `urls.py`. The main area is the Terminal tab, which displays a command-line interface. A yellow box highlights the dropdown menu next to the tab title "1: bash".

```
4 Generated by 'django-admin startproject' using Django 2.2.
5
6 For more information on this file, see
7 https://docs.djangoproject.com/en/2.2/topics/settings/
8
9 For the full list of settings and their values, see
10 https://docs.djangoproject.com/en/2.2/ref/settings/
```

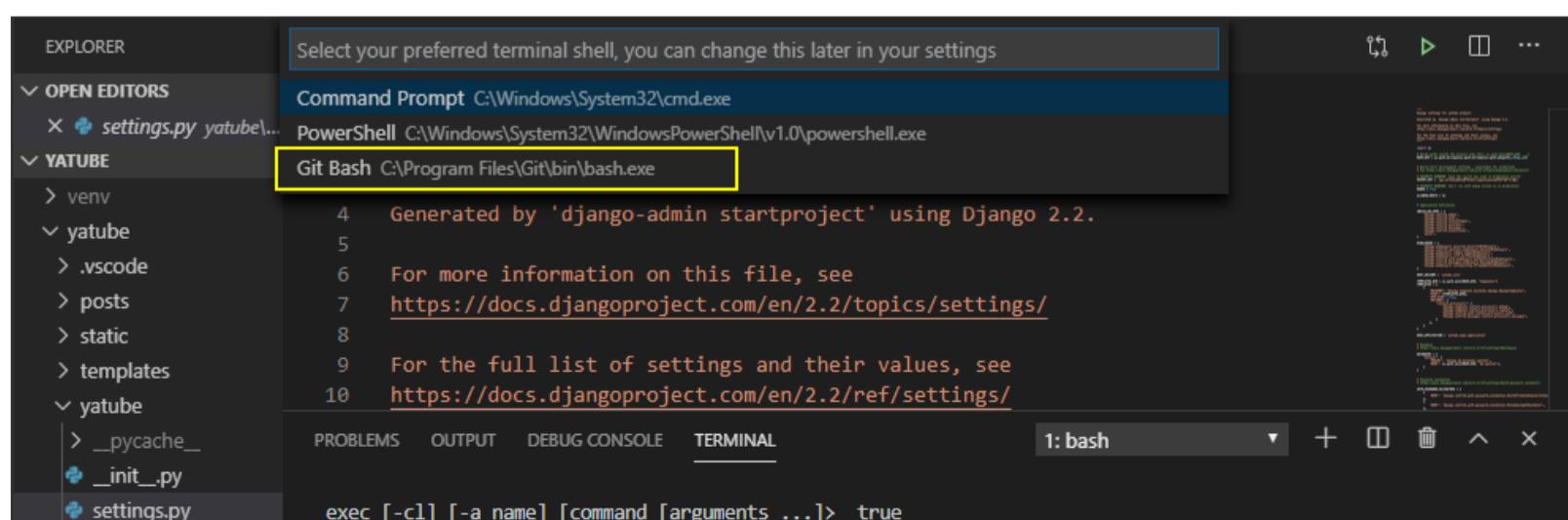
exec [-cl] [-a name] [command [arguments ...]]> true
exit [n] type [-afptP] name [name ...]
`_init_.py`

Если вдруг у вас включен какой-то другой терминал (на Windows, скорее всего, включится *powershell*) — настройте автоматический запуск *bash*, именно в нём мы будем работать на протяжении всего курса.

В выпадающем окне выберите **Select default shell**



В открывшейся панели выберите "Git Bash"

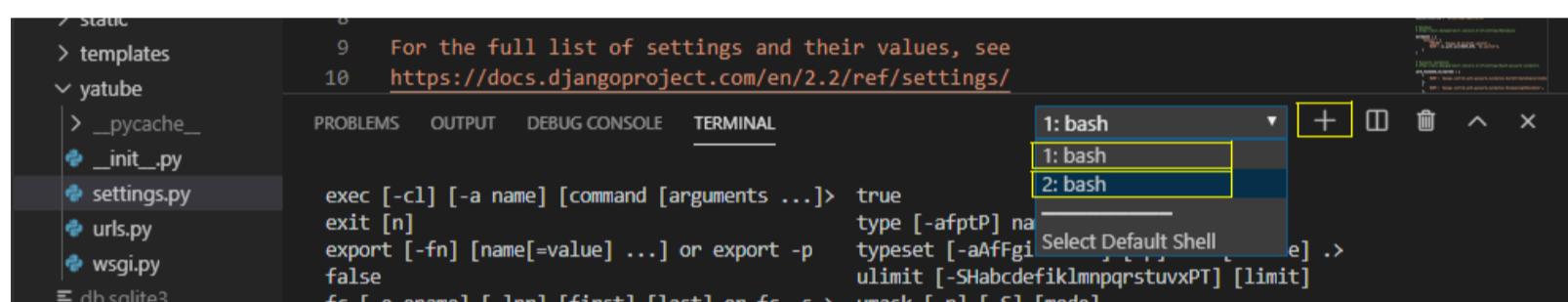


Готово. Теперь при запуске терминала будет автоматически запускаться Git Bash.

Все дальнейшие команды и примеры будут даны именно для Git Bash, запущенном в VSC.

В других терминалах какие-то команды могут не сработать, а если Git Bash запустить не в VSC, а в отдельном окне — ответы терминала могут отличаться от ожидаемых (но всё будет работать).

VSC позволяет запустить несколько терминалов одновременно: кнопка + в интерфейсе терминала откроет ещё одно окно, а через выпадающее меню можно переключаться между терминалами.



Создание виртуального окружения

Бывает, что программист работает сразу над несколькими проектами. Одному проекту нужна старая версия библиотеки, а для другого хочется установить версию посвежее. Каждому проекту нужен уникальный набор библиотек. Для решения этой проблемы в мире Python были созданы «виртуальные окружения», определяющие место, где хранится определённый проект и его библиотеки. Это такие «изолированные территории», где для каждого проекта можно установить собственные правила. Теперь можно одновременно работать со множеством проектов, не переживая, что их зависимости будут мешать друг другу.

Сейчас мы создадим виртуальное окружение и поставим в него Django.

Запустите редактор Visual Studio Code и через меню «Файл» / «Открыть директорию» откройте папку poemnetwork. Запустите терминал в VSC, удостоверьтесь, что вы работаете из директории poemnetwork — и выполните команду:

```
$ python -m venv venv
# Python, запусти модуль (-m) venv, он установит виртуальное окружение.
# Назови это виртуальное окружение "venv", чтобы его можно было вызвать по имени.
```

```
# Имя можно задать любое, мы в примерах будем использовать "venv".
```

После выполнения этой команды в директории проекта появится папка `venv` (от *virtual environment*, «виртуальное окружение»), в которой хранятся служебные файлы виртуального окружения. Там же будут сохранены все зависимости проекта.

Запуск виртуального окружения проекта

Каждый раз перед началом работы с проектом нужно запускать виртуальное окружение. При активированном окружении проект будет работать внутри собственного «загончика», в котором ему будет доступна собственная версия Python и те зависимости, которые установлены именно для этого проекта.

В терминале убедитесь, что вы находитесь в корневой директории проекта и активируйте виртуальное окружение.

В Windows для запуска виртуального окружения выполните такую команду:

```
# выполнить инструкции из файла activate во вложенной папке venv/Scripts  
$ source venv/Scripts/activate
```

В macOS или Linux виртуальное окружение запускается так:

```
# выполнить инструкции из файла activate во вложенной папке venv/bin  
$ source venv/bin/activate
```

При работе с проектом эти команды будут вам нужны постоянно. Скопируйте в блокнотик, будет удобно.

В терминале появится уведомление о том, что вы работаете в виртуальном окружении: строка (`venv`) будет предварять все команды.

```
# строка для ввода теперь будет выглядеть так,  
# и именно такой вариант будет в примерах кода:  
(venv) $
```

```
# но, в зависимости от настроек вашей системы, строка ввода может выглядеть  
чуть иначе.  
# всё нормально, так тоже работает:  
(venv)  
username@computer-name /directory-name (master)  
$
```

Магия виртуального окружения работает таким образом, что пользователю нужно его однократно активировать — и с выбранным окружением будут связаны все команды.

Все дальнейшие команды в терминале надо выполнять с активированным виртуальным окружением. Команды будут выглядеть так: (venv) \$ команда

Остановить работу виртуального окружения можно командой:

```
(venv) $ deactivate
```

Установка первого пакета

Установите через `pip` первый пакет в виртуальное окружение (перед выполнением команды убедитесь, что виртуальное окружение запущено и что вы работаете из корневой директории проекта)

```
# менеджер пакетов pip, поставь мне, пожалуйста, Django версии 2.2
(venv) $ pip install Django==2.2
```

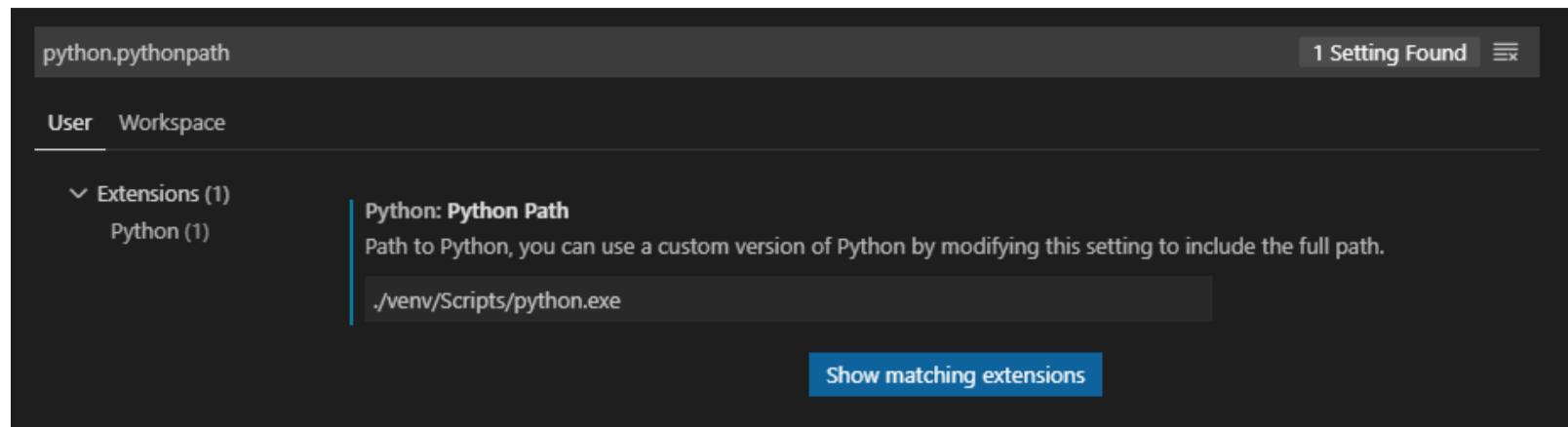
По этой команде запустится менеджер пакетов `pip`, он обратится к индексу пакетов на <https://pypi.org/>, найдет нужный пакет — *Django версии 2.2*, прочитает список его зависимостей, скачает нужные пакеты и установит их в виртуальное окружение проекта.

Версия 2.2 имеет статус LTS (*Long Term Support*, «расширенная поддержка») и будет поддерживаться до апреля 2022 года. Django поддерживает обратную совместимость, поэтому не пугайтесь, когда выходит новая версия: все ваши проекты на прежних версиях Django будут нормально работать.

Настройка редактора

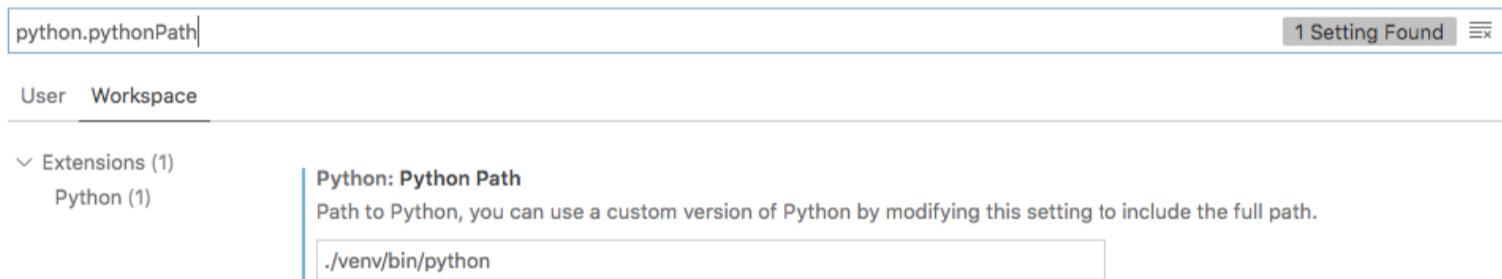
Остался последний шаг: настроить привязку *Python* к текущему проекту в редакторе. Зайдите в настройки VSC: **File** (или **Code** на macOS) > **Preferences** > **Settings**. Переключитесь на закладку *Workspace*. В строке поиска введите имя ключа конфигурации `python.pythonPath` и укажите значение:

- **Для Windows** `./venv/Scripts/python.exe` или полный путь



к `python.exe`, который расположен в папке `venv/Scripts`, должно получиться что-то типа: `d:/Dev/poemnetwork/venv/Scripts/python.exe`

Для macOS и Linux `./venv/bin/python` или полный путь к файлу `python` в директории `venv`



Готово! У вас установлено окружение для работы проектом `poemnetwork`.

При настройке других проектов смело повторяйте все шаги этого урока. Единственное отличие может быть при установке новых пакетов, если вы будете создавать новый проект без Django. Заново устанавливать редактор VSC тоже не потребуется.

02_Создание основы проекта

Мы уже создали виртуальное окружение для проекта, теперь создадим его файловую структуру. Этот процесс автоматизирован, а структура проекта стандартизирована.

Процесс автоматического создания основы проекта называется *scaffolding*. Этот термин не имеет устоявшегося аналога в русском языке, но часто используется в англоязычной документации. Литературный перевод звучит

как «возведение строительных лесов». Применимельно к коду можно перевести как «автоматическое создание основы проекта».

Итак, возьмёмся за *scaffolding*. В командной строке убедитесь, что вы находитесь в папке проекта `poemnetwork` (или перейдите в неё). Виртуальное окружение должно быть активировано: строка ввода в терминале должна начинаться с `(venv)`. Если всё ок — запустите в терминале команду создания базовой структуры проекта:

```
(venv) $ django-admin startproject poemnetwork
```

После выполнения этой команды у вас на диске появится такая структура директорий и файлов:

```
Poemnotes //папка проекта
├── poemnotes //основная рабочая папка с кодом проекта
│   ├── manage.py
│   └── poemnotes //папка с настройками проекта
│       ├── __init__.py
│       ├── settings.py
│       ├── urls.py
│       └── wsgi.py
└── venv //папка виртуального окружения
├── README.md
└── .gitignore
```

В корневую директорию проекта `Yatube` вложены папки `yatube` и `venv`. Откройте в редакторе файл `README.md`: это файл для описания проекта. Напишите в нём, что это ваш учебный проект для Практикума.

Содержимое директорий проекта:

- В директории `Poemnotes/poemnotes` лежат файлы с кодом вашего проекта
- `Poemnotes/poemnotes/manage.py` — файл управления Django-проектом и его запуском из командной строки
- Пустой файл `Poemnotes/poemnotes/poemnotes/__init__.py` объявляет директорию Python-пакетом. По наличию такого файла Python поймёт, что функции из этой директории можно импортировать, например с помощью команды `import poemnotes.urls`

- В файле `Poemnotes/poemnotes/poemnotes/settings.py` хранятся все настройки проекта. При развёртывании проекта автоматически устанавливаются дефолтные настройки, а в ходе работы настройки изменяют или дополняют. Обычно этот файл называют «конфиг». Более педантичные программисты говорят «файл настроек» или «файл конфигурации проекта», но таких программистов мало.
- Шаблоны для *URL Mapping* сохранены в файле `Poemnotes/poemnotes/poemnotes/urls.py`
- `Poemnotes/poemnotes/poemnotes/wsgi.py` — это файл конфигурации WSGI-сервера, он пригодится при настройке вашего сайта на сервере.

Запуск сайта

Файл `manage.py` — это оболочка для управления модулями Django. В дальнейшем мы будем через него добавлять новых администраторов (`python manage.py createsuperuser`), запускать проект (`python manage.py runserver`), создавать новые приложения (`python manage.py startapp`). Полный список доступных команд можно увидеть, запустив в терминале команду вызова справки: `python manage.py --help`.

Дальнейшие команды мы будем выполнять из папки с кодом проекта `Poemnotes/poemnotes`.

Перейдите в неё и запустите сервер проекта:

```
(venv) $ cd poemnotes
(venv) $ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...
```

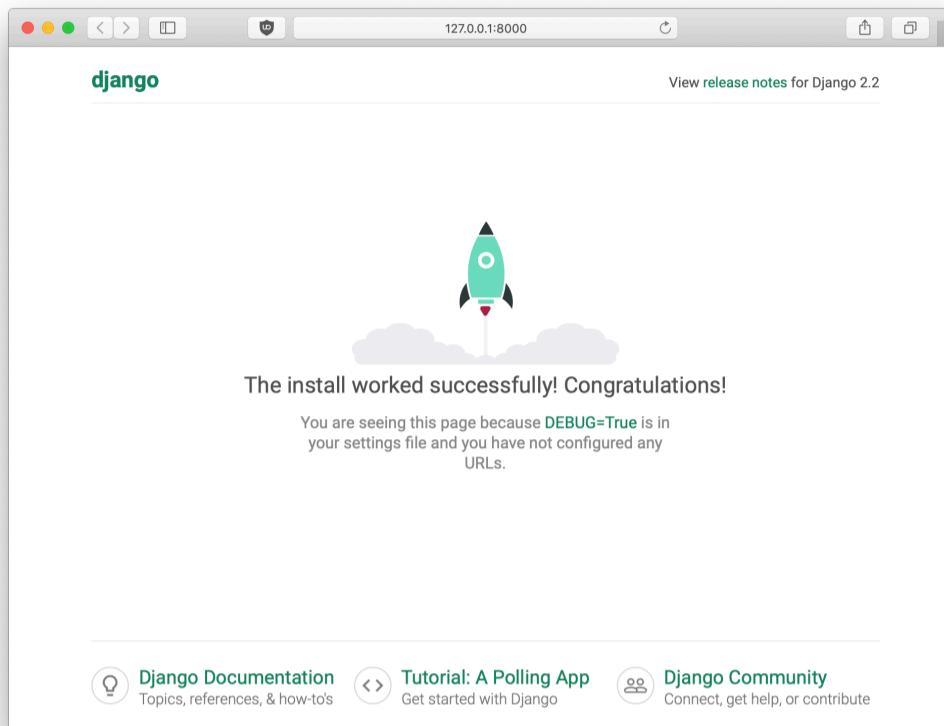
```
System check identified no issues (0 silenced).
```

```
You have 17 unapplied migration(s). Your project may not work properly until
you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
```

```
September 10, 2019 - 15:03:24
Django version 2.2, using settings 'poemnotes.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Объявив, что разработческий сервер доступен по адресу <http://127.0.0.1:8000/>, Django указывает вам на 17 каких-то недоделок, **unapplied**

migration(s). О них мы ещё поговорим. А сейчас перейдите по ссылке <http://127.0.0.1:8000/> и насладитесь успехом:



Ваш сайт на Django стартовал! Вы ещё ничему его не научили, но он работает и готов к взлёту.

Когда понадобится остановить сервер — закройте окно терминала или нажмите Control+C.

03_Создание первого приложения

В терминологии Django проект состоит из одного или нескольких приложений (*application*, сокращенно *app*). Приложение — это специальным образом оформленный Python-пакет, имеющий стандартную структуру.

Проект Poemnotes — платформа для публикаций, блог. Нужно дать пользователям публиковать записи. По-английски публикацию на сайте часто называют *post*. Создадим приложение для управления публикациями. Всю файловую структуру приложения в Django можно создать одной командой из консоли.

Выполните команду из директории Poemnotes/poemnotes

```
(venv) poemnotes $ python manage.py startapp posts
```

В папке с кодом проекта появится директория `posts`, включающая стандартный для любого приложения набор директорий и файлов:

```
posts
├── __init__.py
├── admin.py
├── apps.py
└── migrations
    └── __init__.py
├── models.py
└── tests.py
└── views.py
```

Django берёт большой кусок работы на себя: вам не надо тратить время на рутину.

04_Django ORM

Django-проект должен хранить массу данных: посты пользователей, информацию о самих пользователях, большое количество технической информации. Пора организовать взаимодействие кода проекта и базы данных.

Классы описывают новые типы объектов и позволяют создавать экземпляры таких объектов.

Записи в базах данных тоже описывают объекты — наборы свойств, которыми можно управлять. Вы уже пережили множество прекрасных мгновений, работая со SQL-запросами.

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

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. Вот класс, описывающий данные для нашего проекта **Poemnotes**:

Класс: Post

Свойства:

- Текст публикации
- Дата публикации
- Автор

```
# так мы записывали обычные классы
class Post:
    def __init__(self, text, pub_date, author):
        self.text = text
        self.pub_date = pub_date
        self.author = author
```

Классы, с которыми работает ORM, называются **модели**. Для них в Django есть специальный синтаксис.

так выглядит синтаксис модели – класса, с которым работает ORM

```
class Post(models.Model): # класс Post, наследник класса Model из библиотеки
    models
        # свойство text типа TextField
        text = models.TextField()

        # свойство pub_date типа DateTimeField, текст "date published" это заголовок
        # поля в интерфейсе администратора. auto_now_add говорит, что при создании
        # новой записи автоматически будет подставлено текущее время и дата
        pub_date = models.DateTimeField("date published", auto_now_add=True)

        # свойство author типа ForeignKey, ссылка на модель User
        author = models.ForeignKey(User, on_delete=models.CASCADE)
```

Для всех **моделей** Django ORM создаст в БД таблицы. Описанные через синтаксис `имя_свойства = models.тип_данных()` свойства модели определят названия и типы данных в колонках таблицы БД.

Так, для модели Post в базе данных будет создана таблица с колонками `text`, `pub_date` и `author`, причём в колонке `author` должны быть указаны Primary Key записей из таблицы `User`.

Магия здесь 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, введенных в веб-формы
- Предоставит возможность изменения таблиц в БД (**миграций**, о них позже)
- 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

Впечатляющий список.

Подробный взгляд на модель

Модель, соответствующая нашей задаче, полностью выглядит так:

```
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("date published", auto_now_add=True)
    author = models.ForeignKey(User, on_delete=models.CASCADE)
```

Первые строки импортируют нужные модули.

Всё, что относится к моделям, импортируется из модуля `db` в классе `models`

```
# из модуля db импортируем класс models
from django.db import models
# из модуля auth импортируем функцию get_user_model
from django.contrib.auth import get_user_model
```

В проекте **Poemnotes** мы дадим пользователям возможность регистрироваться и создавать свои страницы, и нам нужен инструмент для создания и администрирования аккаунтов. В Django встроена работа с пользователями. Для управления ими создана специальная модель `User`, и мы импортируем её. Официальная документация рекомендует обращаться к модели `User` через функцию `get_user_model`. Следуем этой рекомендации:

```
User = get_user_model()
```

Описание модели начинается с объявления: класс `Post` — это наследник класса **Model** из модуля **models**.

Класс `Model` нужен для того, чтобы от него наследовались *модели*, классы, обрабатывающие данные. У класса `Model` есть множество предустановленных свойств и методов, обеспечивающих работу с БД.

Свойства модели связаны со столбцами таблицы в БД, а методы превращаются в запросы.

```
class Post(models.Model):
    text = models.TextField()
    pub_date = models.DateTimeField("date published", auto_now_add=True)
    author = models.ForeignKey(User, on_delete=models.CASCADE)
```

Для свойств моделей указывают типы данных, соответствующие типам данных в БД.

В коде модели `Post` мы применили:

- **TextField** — поле для хранения произвольного текста;
- **DateTimeField** — поле для хранения даты и времени. Существуют похожие типы: для хранения даты **DateField**, промежутка времени **DurationField**, и 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.

Добавление модели в проект

Время кодить. Разместим код модели *Post* в проекте. В *Django* код моделей хранят в файлах `models.py`. Обычно такой файл создают в каждом приложении (*app*) проекта.

Вы создали приложение *Posts*, в его директории появился файл `posts/models.py`. По умолчанию в этом файле прописан только импорт модуля `models`:

```
from django.db import models

# Create your models here.
```

Добавьте в файл код модели *Post*. Теперь `models.py` должен выглядеть так:

```
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("date published", auto_now_add=True)
    author = models.ForeignKey(User, on_delete=models.CASCADE,
related_name="posts")
```

Обратите внимание на поле **author**. Оно ссылается на автора поста, на модель **User**, и для этого поля указано свойство `related_name="posts"`.

Тут снова начинается магия: в каждом объекте модели **User** автоматически будет создано свойство с таким же названием (`posts`), и в нём будут

храняться ссылки на все объекты модели **Post**, которые ссылаются на объект **User**.

На практике это означает, что в объекте записи есть поле `author`, в котором хранится ссылка на автора(например, **admin**), а в объекте пользователя **admin** появилось поле `posts`, в котором хранятся ссылки на все посты этого автора. И теперь можно получить список постов автора, обратившись к его свойству `posts`

В следующем уроке вы добавите модель в базу данных проекта.

05_Работа с миграциями

В прошлом уроке мы создали модель **Post** в приложении **Posts**. Пока что эта модель — просто код, связи с базой данных у неё нет. Да и самой БД тоже нет. Сейчас мы создадим её и подключим к проекту.

В файле `poemnotes/settings.py` есть опция `DATABASES`:

```
# Database
# https://docs.djangoproject.com/en/2.2/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

В свойстве `ENGINE` указано, какой драйвер баз данных использует проект. Django умеет работать с разными базами данных, в нашем проекте это будет **SQLite**. Она была автоматически установлена на вашем компьютере вместе с Python.

Свойство `NAME` указывает, что файл с базой данных будет создан в головной директории проекта и будет называться `db.sqlite3`.

При первом запуске сервера Django пытался прочитать содержимое базы данных, не нашёл её и предупредил, что БД не подготовлена для работы. Мы тогда собирались поговорить о семнадцати «недоделках». Пришло время сдержать обещание: теперь вы понимаете, что означает системное сообщение "*You have N unapplied migration(s)*".

Оно информирует, что файл БД пуст или в нём не созданы необходимые таблицы. С этим нужно что-то делать.

В командной строке из директории Poemnotes/poemnotes выполните команду (venv) \$ python manage.py runserver:

```
(venv) $ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 17 unapplied migration(s). Your project may not work properly until
you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.

October 01, 2019 - 10:59:42
Django version 2.2, using settings 'poemnotes.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Миграция данных

В Django изначально есть некоторое количество встроенных моделей, которые нужны для работы системы, но таблицы для них ещё не созданы. Можно создать таблицы «ручками»: написать SQL-запрос на создание таблиц, указать имена и типы полей для каждой таблицы, etc.

А можно командой из терминала запустить **миграцию** — автоматический процесс обновления базы данных на основании определённых правил.

При запуске миграции Django проверит все модели в коде и сопоставит их с существующими таблицами в БД.

Если для какой-то модели таблицы нет, Django создаст её. А если таблица не соответствует модели, то будет изменена и согласована с этой моделью.

Добавление приложения в проект

Пока что Django не знает о приложении **Posts**: каждый новый app нужно зарегистрировать в проекте, иначе при запуске миграций Django не заглянет директорию приложения в поисках модели.

Отредактируйте файл `poemnotes/settings.py` — найдите в файле переменную `INSTALLED_APPS` и впишите в неё имя приложения **Posts**. Должно получиться примерно так:

```
INSTALLED_APPS = [
    'posts', # наше приложение posts**
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Порядок перечисления приложений имеет значение — в некоторых особых случаях. Сейчас для простоты будем добавлять все свои приложения в начало списка.

Создание и запуск скриптов миграции

Теперь Django знает о приложении **Posts**, можно создавать миграцию. Из директории `Poemnotes/poemnotes` запустите команду создания скрипта миграций `makemigrations`, получите следующий ответ:

```
(venv) $ python manage.py makemigrations
Migrations for 'posts':
  posts/migrations/0001_initial.py
    - Create model Post
```

В приложении **Posts** в папке `migrations` будет создан файл со скриптом миграции: в нём сохранено описание моделей, найденных в этом приложении. Для встроенных моделей такие скрипты были созданы при установке Django.

Теперь нужно запустить все миграции. Выполните команду `migrate`:

```
(venv) $ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, posts, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
```

```
Applying auth.0005_alter_user_last_login_null... OK
Applying auth.0006_require_contenttypes_0002... OK
Applying auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying auth.0009_alter_user_last_name_max_length... OK
Applying auth.0010_alter_group_name_max_length... OK
Applying auth.0011_update_proxy_permissions... OK
Applying posts.0001_initial... OK
Applying sessions` .0001_initial... OK
```

OK обозначает, что операция успешно завершена. Вот они, те 17 исправлений, которых недоставало при запуске проекта.

Теперь к проекту **Poemnotes** подключена база данных, в неё добавлены таблицы служебных приложений и создана таблица для приложения **Posts**. Создали модель, запустили миграцию — и всё готово для сохранения постов пользователей.

06_Создание суперпользователя

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

- `admin` — создаёт интерфейс администратора сайта,
- `auth` — управляет работой с пользователями.

Именно эти приложения позволяют нам создать учётную запись администратора сайта и авторизоваться на сайте. При создании администратора мы дадим ему максимум прав. Такие аккаунты в Django называются «суперпользователи» (`superuser`).

Под этой учётной записью вы будете управлять сайтом.

Для создания суперпользователя выполните команду:

```
(venv) $ python manage.py createsuperuser
Username (leave blank to use 'user'): # придумайте логин (например -- admin)
Email address: # укажите почту
Password: # придумайте пароль
Password (again): # повторите пароль
Superuser created successfully.
```

Если вы укажете слишком простой пароль (например, 111), Django предложит усложнить его:

```
(venv) $ python manage.py createsuperuser
Username (leave blank to use 'user'): admin
```

```
Email address:   
Password:   
Password (again):   
This password is too short. It must contain at least 8 characters.  
This password is too common.  
This password is entirely numeric.  
Bypass password validation and create user anyway? [y/N]: ^C  
Operation cancelled.
```

Если вы забудете имя пользователя или логин, то, при наличии доступа к серверу, вы всегда можете создать нового суперпользователя.

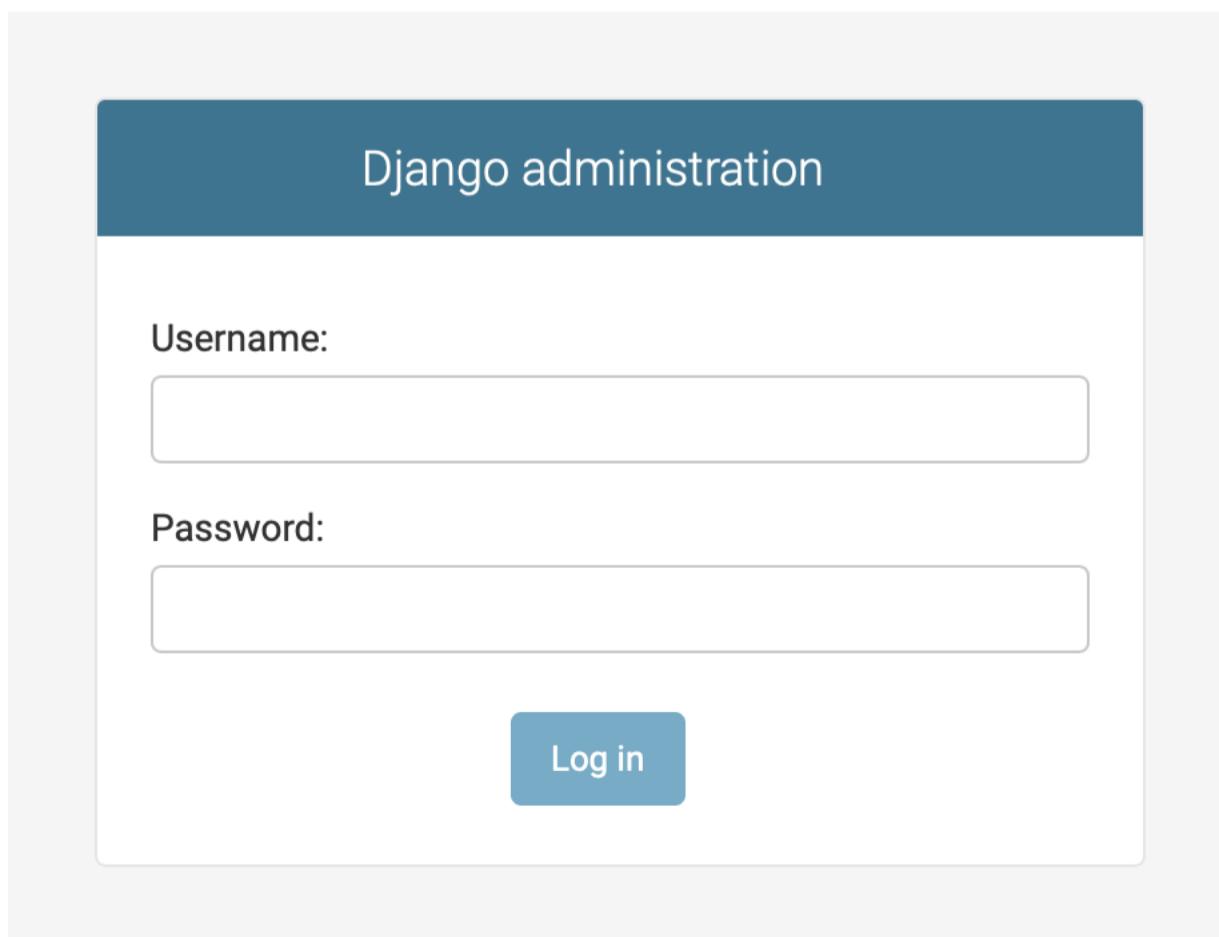
Работа с интерфейсом администратора сайта

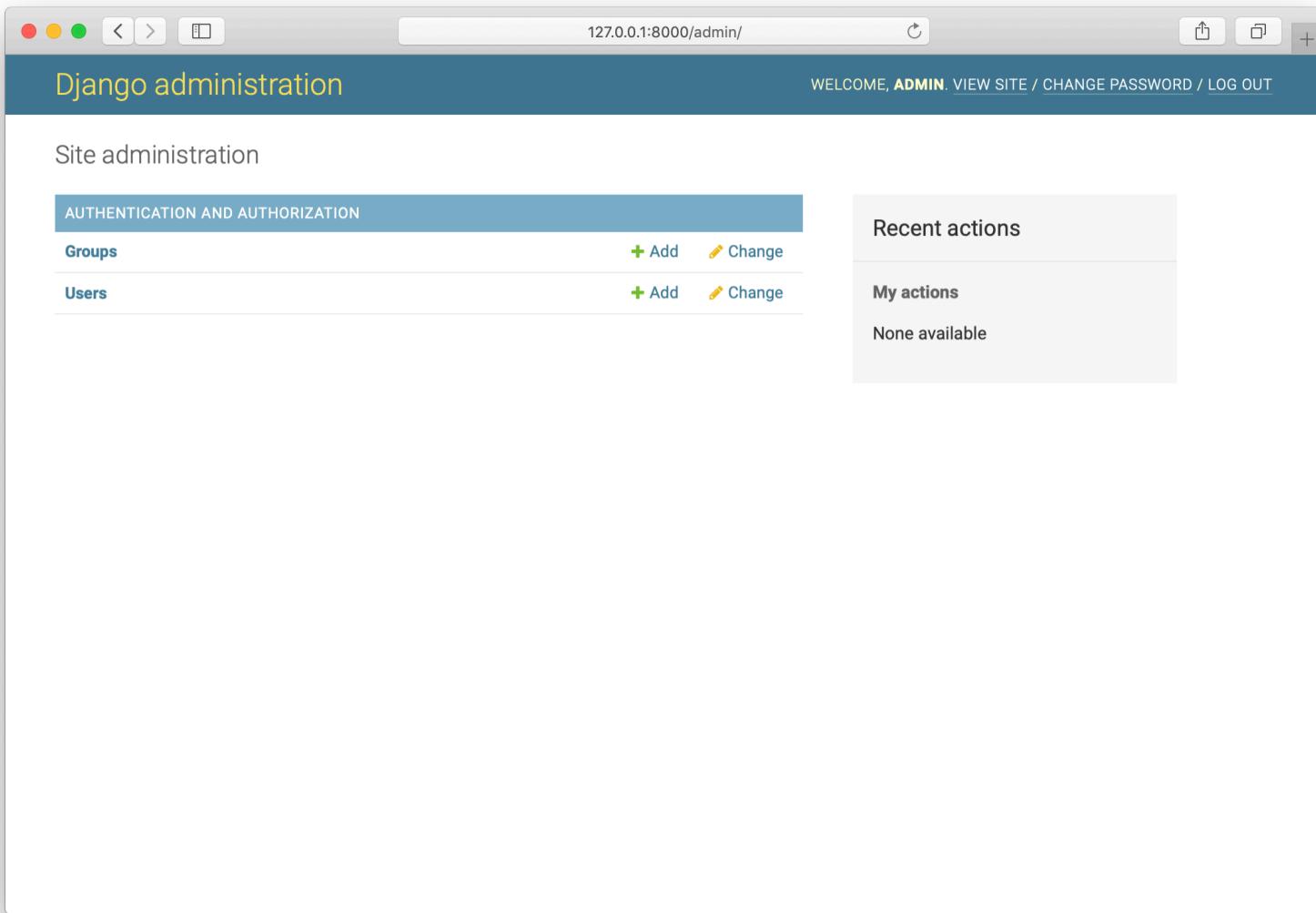
Административный интерфейс сайта на жаргоне называют «админка».

Запустите сайт:

```
(venv) $ python manage.py runserver  
Watching for file changes with StatReloader  
Performing system checks...  
  
System check identified no issues (0 silenced).  
October 01, 2019 - 14:40:27  
Django version 2.2, using settings 'yatube.settings'  
Starting development server at http://127.0.0.1:8000/  
Quit the server with CONTROL-C.
```

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





Введите имя и пароль суперпользователя.

Добро пожаловать! Вы попали в административный раздел сайта:

Интерфейс админки сейчас на английском, а для управления пока доступен только раздел "AUTHENTICATION AND AUTHORIZATION".

В следующем уроке мы выведем на эту страницу интерфейс управления приложением *posts*.

07_Регистрация модели Post в админке

Добавим в админку интерфейс для приложения Posts, чтобы администратор сайта получил возможность управлять публикациями.

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 Post в интерфейс администратора, её надо зарегистрировать в файле `posts/admin.py`.

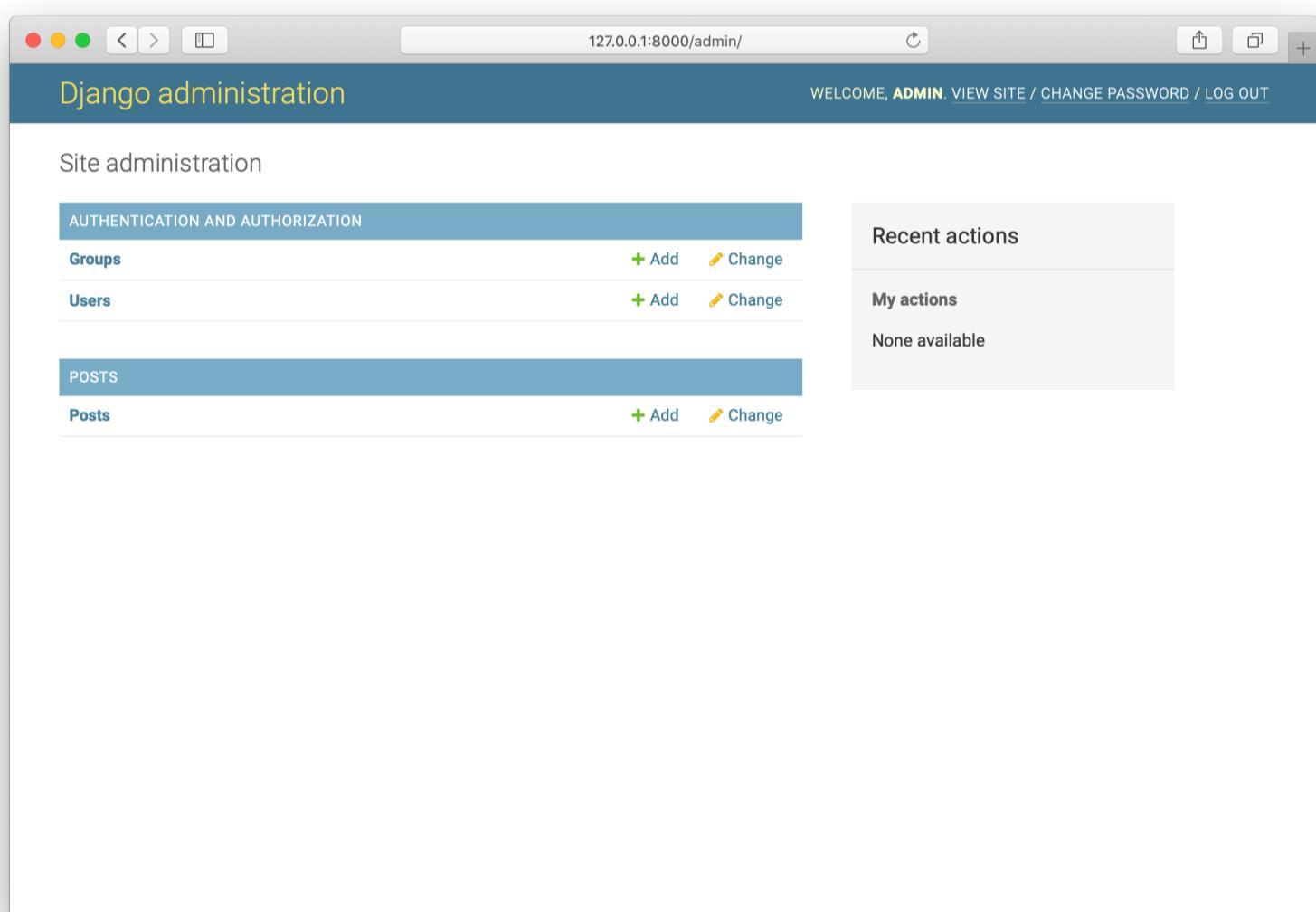
Пока что файл `posts/admin.py` пуст:

```
from django.contrib import admin  
  
# Register your models here.
```

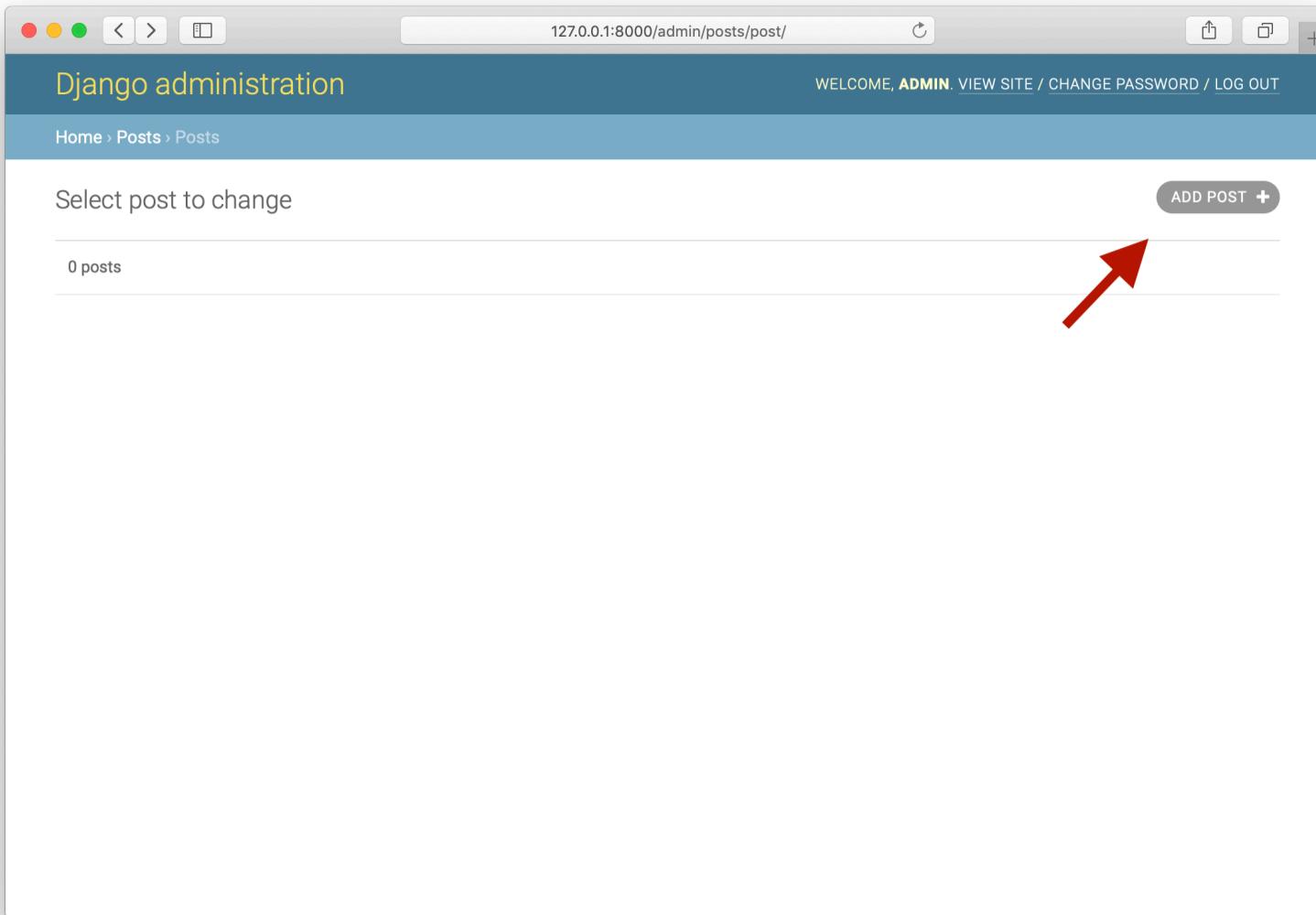
Добавьте в него такой код:

```
from django.contrib import admin  
# из файла models импортируем модель Post  
from .models import Post  
  
admin.site.register(Post)
```

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



Все модели, зарегистрированные в `admin.py` определённого приложения, в админке будут отображаться в разделе этого приложения (при этом сама модель этому приложению может не принадлежать). Например, если зарегистрировать модель `Post` в `admin.py` приложения `auth` – подраздел `Posts` «переедет» в раздел `Authentication and authorization`.

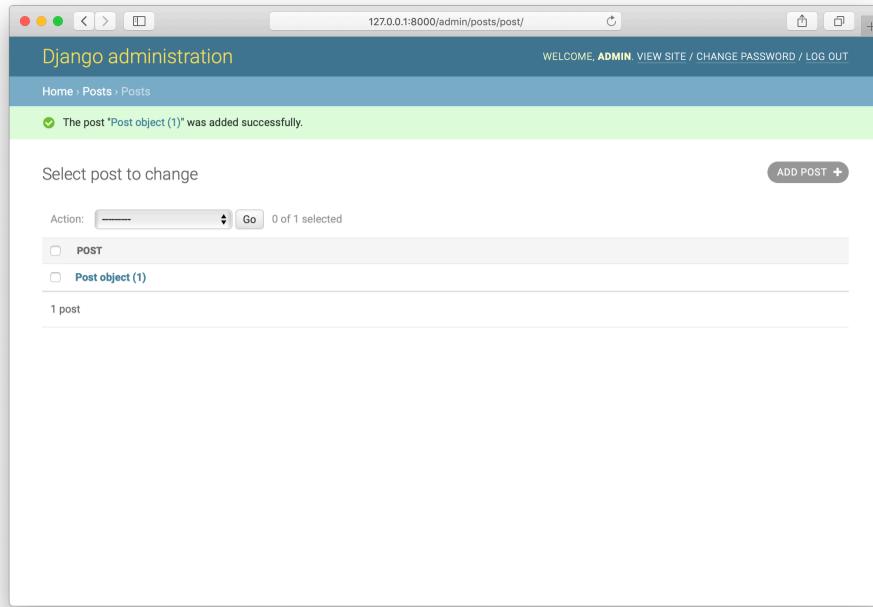


Перейдите в раздел Posts: там подготовлено место для списка постов и есть кнопка для создания новой публикации. При нажатии откроется форма для создания поста.

Создайте новый пост. Пишите, что хотите; в нашем примере всего два слова: «Буду краток».

С точки зрения разработчика создание нового поста — это создание нового объекта (экземпляра класса) Post. Но класс Post — это модель, так что свойства, переданные в экземпляр этого класса, станут строчками в таблице БД, связанной с моделью.

Поле «Автор» обязательно для заполнения. Если не указать автора, то система выдаст сообщение об ошибке. Но если сделать всё правильно, вы создадите на сайте новую запись:



В перечне публикаций строка Post object(1) выглядит некрасиво и неинформативно. Если таких записей будут десятки, то разобраться в постах будет невозможно. Сейчас мы настроим отображение списка получше.

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. Полный список параметров есть в документации.

В файле `posts/admin.py` создайте класс `PostAdmin`, наследующийся от `admin.ModelAdmin`, и зарегистрируйте его как источник конфигурации для модели `Post`.

Теперь файл `posts/admin.py` должен выглядеть так:

```
from django.contrib import admin
from .models import Post

class PostAdmin(admin.ModelAdmin):
    # перечисляем поля, которые должны отображаться в админке
    list_display = ("text", "pub_date", "author")
    # добавляем интерфейс для поиска по тексту постов
    search_fields = ("text",)
    # добавляем возможность фильтрации по дате
    list_filter = ("pub_date",)

# при регистрации модели Post источником конфигурации для неё назначаем класс
# PostAdmin
admin.site.register(Post, PostAdmin)
```

Столбцы таблицы в админке изменятся, добавится поле поиска и фильтрация постов по дате публикации. Django понимает, что свойство модели `pub_date` — это дата, и предложит несколько вариантов фильтрации.

Сохраните код, обновите страницу и посмотрите, что получилось:

| TEXT | DATE PUBLISHED | AUTHOR |
|--|-------------------------|--------|
| <input type="checkbox"/> Привет я текст! | Oct. 1, 2019, 3:04 p.m. | admin |

Свойства, которые мы настроили:

- **list_display** — перечень свойств модели, которые мы хотим показать в интерфейсе. Если это свойство не указано — будет отображаться строка Имя_модели(идентификатор), как было с записью `Post(1)`.
- **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.

Помимо этих параметров, есть много других: например, для управления порядком отображения или количеством элементов на экране.

Пустое поле

В колонку `text` сейчас выводится содержимое поля `text` каждого поста. Этот текст — ссылка на форму просмотра и редактирования записи. Но что делать, если у модели нет описательного поля? Вдруг админ магазина не заполнит название товара или пост в блоге состоит только из картинки без подписи? Поле в админке окажется пустым, и у администратора не будет ссылки на страницу редактирования записи.

Есть несколько способов обыграть эту ситуацию. Прежде всего, у каждой модели существует специальное свойство `pk` (сокращение от *Primary Key*, уникальный идентификатор записи в базе данных). Можно вывести *Primary Key* в интерфейс администратора:

```
class PostAdmin(admin.ModelAdmin):
    # добавим в начало столбец pk
    list_display = ("pk", "text", "pub_date", "author")
    search_fields = ("text",)
    list_filter = ("pub_date",)
```

В списке постов появится поле с номером записи в базе, каждая запись в админке получит уникальный ID.

Есть и другой способ: вместо пустого поля в строке можно подставлять какое-нибудь дефолтное значение, указав его в конфигурации модели:

```
class PostAdmin(admin.ModelAdmin):
    list_display = ("pk", "text", "pub_date", "author")
    search_fields = ("text",)
    list_filter = ("pub_date",)
    empty_value_display = "-пусто-" # это свойство сработает для всех колонок:
    где пусто - там будет эта строка
```

Теперь слово «пусто» станет ссылкой, по которой админ может кликнуть, чтобы перейти на страницу редактирования поста.

08_Загрузка тестовых данных

При разработке проекта хорошо работать с базой, наполненной какими-то реальными данными. Мы подготовили для вас базу данных с тестовой информацией.

Скачайте файл, извлеките его из архива и замените им ваш файл `db.sqlite3`.

Вместо вашей БД у вас будет база, заполненная тестовыми постами.

<https://code.s3.yandex.net/backend-developer/learning-materials/db.sqlite3.zip>

После замены файла создайте суперпользователя заново, ведь информация о нём была удалена вместе с прежней базой, а в новой его нет.

Зайдите в раздел администратора: в списке постов вы увидите дневниковые записи Льва Николаевича Толстого за июль 1854 года. «Ковыряль нось и ничего не написаль» — это не про вас.

| PK | TEXT | DATE PUBLISHED | AUTHOR |
|----|---|-------------------------|--------|
| 37 | [Фокшаны.] Еще переходъ до Фокшань, во время которого я ъхалъ съ Монго. Человѣкъ пустой, но съ твердыми, хотя и ложными убѣжденіями. Генерал[у] по этому должно быть случаю, угодно было спрашивать о моемъ здоровье. Свина! К[о]выряль нось и ничего не написаль — вотъ 2 упрека за нын[ѣшній] день. Послѣдній упрекъ становится слишкомъ часть, хотя походъ и можетъ служить въ немъ отчасти извиненiemъ. Отношенія мои съ товарищами становятся такъ пріятны, что мнѣ жалко бросить штабъ. Здоровье кажется (2) лучше. | July 31, 1854, midnight | leo |
| 36 | [Рымник.] Сдѣлалъ верхомъ переходъ до Рымника. [[5]] Старикъ все не кланяется мнѣ. Обѣ вещи эти злять меня. Съ встрѣчавшимися башни-бузуками вѣль себя хорошо. Объяснился съ Крыжановскимъ. Онъ, не знаю зачѣмъ, совѣтуетъ мнѣ прикомандироваться къ казачьей батарѣ; совѣтъ, которому я не послѣду. Желчно спорилъ вечеромъ съ Фриде и Бабарыкинымъ, ругалъ Сержпутовскому и ничего не сдѣлалъ, вотъ 3 упрека, которыя дѣлаю себѣ за нынѣшній день. (3) | July 30, 1854, midnight | leo |
| 35 | Исправленіе мое идетъ прекрасно. Я чувствую, какъ отношенія мои становятся пріятны и легки съ людьми всякаго рода, съ тѣхъ поръ какъ я рѣшился быть скромнымъ и убѣдился въ томъ, что казаться всегда величественнымъ | July 29, 1854, midnight | leo |

09_Проектирование структуры адресов

В бесплатном курсе по Django вы познакомились с последовательностью обработки запросов. Теперь в эту последовательность можно добавить работу с классами и моделями, и алгоритм будет выглядеть так:

- Данные проекта хранятся в БД.
- Для взаимодействия с БД в коде создаются **модели**.
- Пользователь обращается к какой-то странице сайта, Django сверяет запрошенный адрес с шаблонами адресов в файле `urls.py`.

- Каждый шаблон адреса в `urls.py` связан с определённой функцией или классом, которые обрабатывают входящие данные. Такие функции (или классы) называются **View**.
- View обращается к моделям и через них получает необходимые данные из БД. Эти данные View передает в шаблоны (**Template**).
- Данные выводятся в шаблон и генерируется HTML-документ, который возвращается пользователю.

Структуру адресов страниц придумывают ещё на стадии проектирования сайта. Она — исключительно плод фантазии и логики разработчика. Для проекта Poemnotes структура может быть такой:

- `""` — главная страница с лентой новых постов пользователей.
- `"/<имя пользователя>"` — страница с постами пользователя. Например, адресом личной страницы пользователя Лев Толстой (его логин — `leo`) будет `/leo`.
- `"/<имя пользователя>/<pk поста>"` — адрес страницы отдельного поста, где `pk` — идентификатор поста, первичный ключ записи в БД. В адресе мы «вкладываем» идентификатор поста в аккаунт автора. Страница с первой записью Льва будет иметь адрес `/leo/1`, а запись с `pk=3`, которую создал пользователь Антон, получит адрес `/anton/3`. Теперь по ссылке можно понять, какому автору принадлежит определённый пост. Это информативнее и удобнее, чем адреса вида `/post/1` или `post/3` (хотя технически можно сделать и так).

```
urlpatterns = [
# правила для сопоставления шаблонов URL и функций
    path('' , views.index),
    path('user' , views.account),
    path('user/1' , views.user_first),
    path('user/<int:user-id>' , views.user_page),
    path('user/2' , views.user_second),
    path('user/login' , views.login),
    path('user/logout' , views.logout),
]
```

10_Добавление главной страницы

Начальный курс по Django дал вам представление о том, как работает *URL Mapping* и *Views*.

Перечень правил, связывающих URL сайта с view-функциями, содержится в списке `urlpatterns`. В каждом элементе вызывается специальная функция `path()`. Вот знакомый вам пример `urlpatterns`:

```
urlpatterns = [
    # правила для сопоставления шаблонов URL и функций
    path('', views.index),
    path('accounts/sign-up', acc_views.sign_up),
    path('accounts/sign-in', acc_views.sign_in),
    # а вот path() с параметром name
    path('accounts/my-account', acc_views.my_account, name='account'),
]
```

Функция `path()` принимает такие параметры: `path(route, view, name)`

- **route** — шаблон веб-адреса (URL). Получив HTTP-запрос, Django идет по списку `urlpatterns` сверху вниз, пока не найдёт совпадение запрошенного адреса с **route** в одном из вызовов `path()`. Если нет ни одного совпадения, пользователю вернётся сообщение об ошибке 404: «Страница не найдена».
- **view** — это имя view-функции. Если Django найдёт совпадение с шаблоном, то перенаправит вызов в указанную view-функцию.
- **name** — имя для `path()`, к нему можно обратиться из кода, чтобы установить ссылку на страницу сайта.

Есть и дополнительные параметры функции `path()`, о которых можно прочесть в [документации](#).

Генератор проекта создал главный файл `poemnotess/urls.py`, но в нём будут только ссылки на `urls.py` приложений нашего проекта.

Создадим шаблон адреса главной страницы в `urls.py` приложения `Posts` и напишем view-функцию для обработки запроса к этой странице.

В приложении `Posts` создайте файл `posts/urls.py` и добавьте в него код:

```
from django.urls import path
```

```
from . import views

urlpatterns = [
    path("", views.index, name="index")
]
```

В головной файл poemnotes/urls.py добавьте импорт правил из нового файла:

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    # импорт правил из приложения posts
    path("", include("posts.urls")),
    # импорт правил из приложения admin
    path("admin/", admin.site.urls),
]
```

Запросы к БД можно делать прямо из кода Python, для этого в Django ORM есть специальный синтаксис. Вместо того чтобы писать громоздкие запросы, можно одной строкой получить информацию из базы данных.

В файл posts/views.py добавьте views-функцию index:

```
from django.http import HttpResponse

from .models import Post

def index(request):
    # одна строка вместо тысячи слов на SQL
    latest = Post.objects.order_by('-pub_date')[:10]
    # собираем тексты постов в один, разделяя новой строкой
    output = []
    for item in latest:
        output.append(item.text)
    return HttpResponse('\n'.join(output))
```

В функции index переменная latest получает выборку записей модели Post из БД. После имени модели и специальной точки входа .objects указаны условия запроса.

В запросе мы:

- сортируем записи по свойству pub_date по убыванию, от больших значений к меньшим (об этом говорит знак -), то есть новые записи оказываются вверху выборки

- забираем только первые 10 элементов из полученного списка

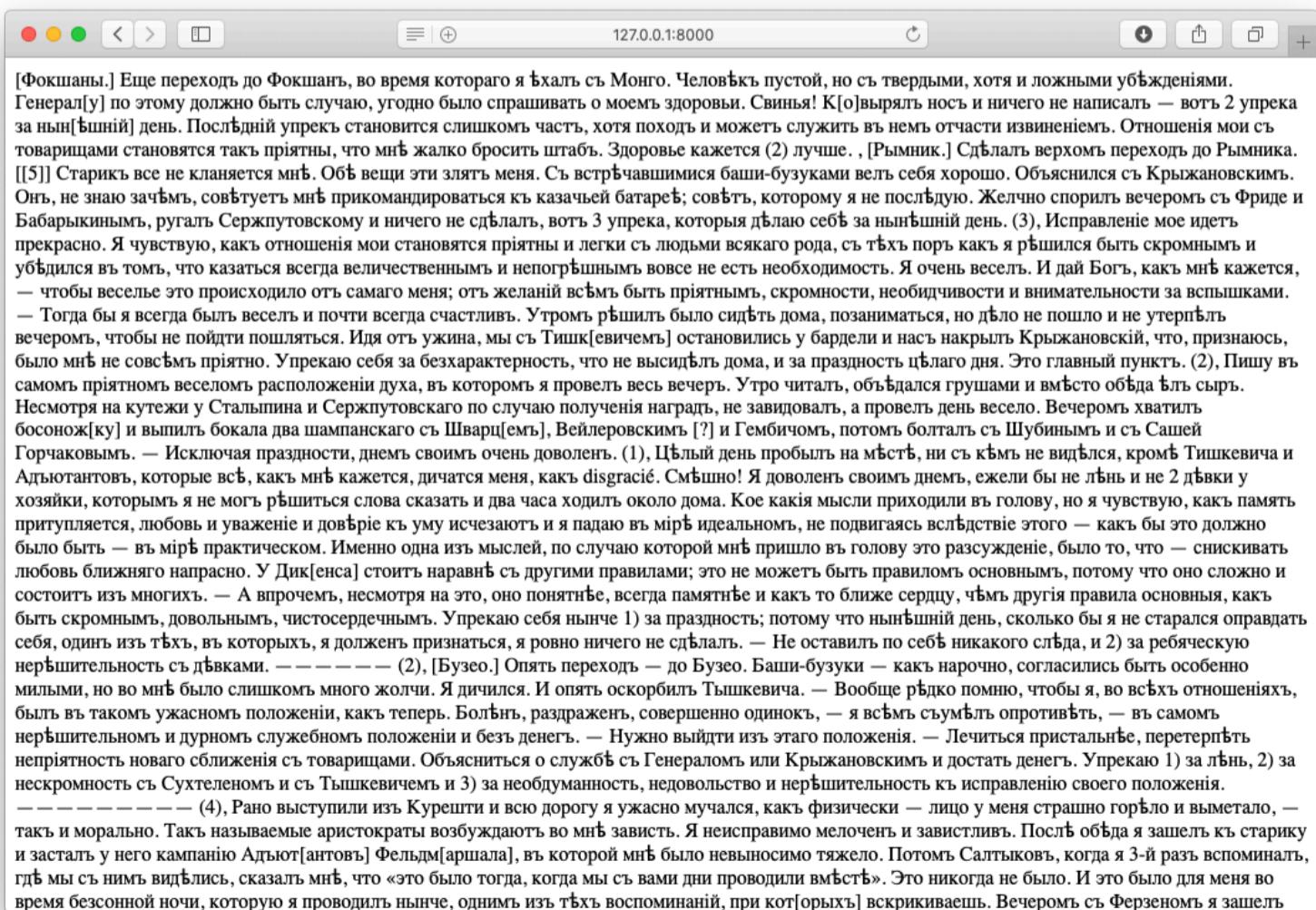
Конструкция написана на безупречном языке Python.

Механизм Django ORM переводит её на SQL:

```
SELECT
    'posts_post'.'id',
    'posts_post'.'text',
    'posts_post'.'pub_date',
    'posts_post'.'author_id'
FROM
    'posts_post'
ORDER BY
    'posts_post'.'pub_date' DESC
LIMIT 10
```

Полученные записи передаются в код как объекты класса Post и в виде списка сохраняются в переменной `latest`. Мы проходим по ним циклом, из каждого объекта достаём значение свойства `text` и добавляем тексты постов в список `output`. Затем возвращаем строку, собранную из элементов этого списка. Это и есть наш ответ на HTTP-запрос.

Сохраните файлы и откройте главную страницу проекта <http://127.0.0.1:8000>



Определенно, это тот же самый текст, что и в админ-зоне. Выглядит не очень, но вы уже сделали большой шаг вперёд: вывели текст из базы данных на главную страницу блога.

Вот ещё несколько примеров, как можно получить данные из базы:

- `Post.objects.all()` — получить все записи модели `Post`
- `Post.objects.get(id=1)` — получить запись модели `Post`, у которой значение поля `id` равно 1. Поскольку поле `id` — это первичный ключ, а Django автоматически создаёт у модели свойство `pk`, то альтернативная запись этого же запроса будет такой: `Post.objects.get(pk=1)`.
- `Post.objects.filter(pub_date__year=1854)` — запрос вернёт объекты, у которых значение года в поле `pub_date` равно 1854. Обратите внимание на синтаксис фильтрации: двойное нижнее подчёркивание между названиями поля и фильтра. Подробнее о функции `filter()` — в [документации](#).
- `Post.objects.filter(text__startswith="Писать не хочется")` — пример фильтра по текстовому полю, он вернёт записи, начинающиеся с указанной в фильтре строки.

11_Шаблоны Django

Вы получили нужные записи из базы данных и вывели их на главную страницу сайта. С технической стороны всё хорошо, но страница выглядит ужасно. Пора подключить HTML-шаблоны — вы знакомились с ними на бесплатном курсе по Django.

Шаблон документа создаётся заранее, это как красивая пустая коробка, обложка без содержимого. В него (например, при вызове функции `render()`) передаются конкретные данные, после чего сформированный HTML-документ отправляется пользователю. Данные в шаблон функция `render()` передаёт словарём (`dict`), обычно этот словарь называют `context`. Значениями элементов этого словаря могут быть любые данные.

Рендеринг шаблона

Страница, созданная на основе шаблонов, должна быть отдана пользователю. Для этого должен пройти специальный процесс, *rendering* (англ. «отрисовка»).

Рендеринг — это превращение исходного кода в результат, который видит пользователь. Например, 3D-мультипликаторы в программах трехмерной графики создают персонажей и эффекты, а потом, после рендеринга исходных файлов, получается видео.

Нам надо из исходников-шаблонов создать для пользователя HTML-документ. Для этого в шаблон «вписываются» переменные, а сама страница может «монтироваться» из нескольких шаблонов.

Результат выполнения функции `render()` — это объект класса `HttpResponse`, объект ответа. В нём хранится HTML-код возвращаемой страницы.

```
from django.shortcuts import render

def my_index(request):
    # какой-то код
    return render(
        request, # первый параметр – это всегда request
        'index.html', # имя шаблона, который нужен для отображения
страницы
        {'title': 'Этот текст встанет в шаблон на место переменной
"title"'}
        # словарь содержит переменные, которые будут переданы в шаблон
    )
```

Переменные

В шаблонах Django есть собственный синтаксис для работы с переменными. При выводе значения переменной её имя указывают в двойных фигурных скобках:

```
{{ variable }}
```

Если при вызове шаблона передать в него переменную по имени `variable`, то вместо конструкции с двойными фигурными скобками будет выведено значение переменной. Вывести переменные в HTML-теги можно так:

```
<h1>
{{ title }}
```

```
</h1>
<p>
    {{ body }}
</p>
```

К элементу словаря, свойству объекта или элементу списка можно обратиться через точечную нотацию:

`{{ var_dict.key }}` – обращение к ключу словаря
`{{ var_instance.attribute }}` – обращение к свойству или методу класса
`{{ var_list.0 }}` – обращение к элементу списка

Процессор шаблонов упрощает обращение к свойствам и ключам объектов. Вам многократно придется пользоваться нотацией через точку. Например если объект **user** содержит свойство **username**, то код в шаблоне будет очень похожим на обращение к объекту в Python: `{{ user.username }}`. Если же объект **user** – это словарь, в котором есть ключ **username**, то в шаблоне все равно надо писать `{{ user.username }}`. Это упрощает работу с переменными, но делает невозможным сложные конструкции, требующие вычислений.

Если у объекта есть метод, то обращаются к нему так же, как к свойству. Обработчик шаблона сам определит, что надо вызвать метод – и опубликует результат вызова. В целях безопасности в шаблонах заблокирована возможность передавать методам параметры.

Теги

Для более сложных конструкций, влияющих на логику исполнения кода, существуют элементы разметки, **теги**. Это не HTML-теги, а совершенно другая сущность. В коде теги шаблонов выделяются конструкциями `{%` и `%}`.

Тег ветвления `{% if %}` очень похож на оператор **if/elif/else** в языке Python. Обратите внимание: у тега **if** есть обязательный закрывающий тег **endif**:

```
{% if user.is_authenticated %}
    Привет, {{ user.username }}.
{% else %}
    Будет здорово, если вы авторизуетесь!
{% endif %}
```

Теги могут получать параметры:

```
{% include "counter.html" %}
```

На место этого тега в шаблон будет встроено содержимое файла **counter.html**.

В Django есть много встроенных тегов, но их можно создать и самостоятельно, мы это сделаем при разработке проекта Poemnotes.

Фильтры

Фильтры нужны для обработки значений переменных или аргументов других тегов. В коде шаблона фильтры присоединяются к фильтруемому значению через символ |:

имя_переменной | фильтр

В Django есть множество встроенных фильтров. Например, фильтр `length` вернёт длину строки или последовательности, переданной в переменной `variable`:

```
 {{ variable | length }}
```

Если в переменную `variable` передать слово «гиппопотомомонстросесквиппедалиофобия», то в шаблон будет выведено число 37, по числу букв в слове.

В бесплатном курсе по Django вы встретились с конструкцией `{{ variable | safe }}` и потестировали её.

Если в шаблоне просто `{{ variable }}` :

Ведите запрос:

Коля, ты где?

Спросить

`<mark>Коля в городе Красноярск</mark>`

Если в шаблоне `variable | safe :`

Введите запрос:

Коля, ты где?

Спросить

Коля в городе Красноярск

Фильтры

МОЖНО

объединять в цепочку:

```
{{ variable | title | truncatewords:4 }}
```

Если переменная `variable` содержит строку Это один маленький шаг для человека и огромный скачок для человечества, то сначала фильтр `title` преобразует регистр букв в формат «каждое слово с заглавной» Это один Маленький Шаг Для Человека И Огромный Скачок Для Человечества, а потом `truncate` обрежет текст до четырёх первых слов. В результате на страницу будет выведено Это один Маленький Шаг.

Фильтр date работает только с объектами типа `date` и `datetime`: он форматирует дату по маске. За основу взят стандарт, принятый в языке программирования PHP.

Если вывести дату без форматирования, то получится что-то вроде `'2019-02-02 00:00:00+00:00'`. Это не лучший вариант. Фильтр `date` позволяет вывести дату в любой форме:

```
{{ pub_date|date:"j.m.Y" }} # выведет 2.02.2019  
{{ pub_date|date:"j F Y" }} # выведет 2 февраля 2019  
{{ pub_date|date:"d.m.y" }} # выведет 02.02.19  
{{ pub_date|date:"d M Y" }} # выведет 02 фев 2019
```

При выводе текстов тоже есть особенности. Пользователю удобно обозначать начало нового абзаца переводом строки, но HTML не понимает такого форматирования и выведет текст сплошным потоком, без разрывов. В Django Templates есть фильтр `linebreaksbr`. Он заменяет символы перевода строки `\n` на HTML-теги `
`.

Комментарии

Django-шаблоны поддерживают комментарии — строки, которые игнорируются при интерпретации кода. Однострочные комментарии записываются между символами `{# и #}`, многострочные — между конструкциями `{% comment %}` и `{% endcomment %}`.

```
{% comment "Опциональный текст, комментарий к комментарию" %}
    <p>Этот кусок шаблона временно отключен {{ create_date|date:"c" }}</p>
{% endcomment %}
```

12_Теги шаблонов Django - наследование и переопределение

Теги Django-шаблонов пишутся между конструкциями `{%` и `%}`.

Теги могут быть одиночные:

```
{% include "footer.html" %}
```

или парные, из открывающего и закрывающего:

```
{% block %}
    тело тега
{% endblock %}
```

Возможны и более сложные варианты: например, у тега **for** есть вложенный тег **empty**. После имени тега могут идти параметры, например `{% if variable %} ... {% endif %}`. К параметрам можно применять фильтры `{% if messages | length >= 100 %}`.

Создадим несложный шаблон и препарируем его:

```
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}The Last Social Media You'll Ever Need{% endblock %}
    | Poemnotes</title>
    <link rel="stylesheet" href="style.css">
</head>

<body>
    {% include "header.html" %}
    <nav id="sidebar">
```

```

{%- block sidebar %}
<ul>
    <li><a href="/">Главная</a></li>
    <li><a href="/about/">0 сайте</a></li>
    <li><a href="/list/">Сообщения</a></li>
</ul>
{%- endblock %}
</nav>

<div id="content">
    {%- block content %}Контент не подвезли :( {%- endblock %}
</div>
{%- include "footer.html" %}
</body>
</html>

```

Обычно подобный шаблон сохраняется в файле `base.html`. Он предоставляет базовую структуру документа.

Тег **block**

Содержимое тега **block** может быть переопределено из другого шаблона. Если значение тега не переопределено, на страницу будет выведено значение, которое предустановлено в шаблоне (это значение может быть и пустым).

После названия тега `block` указывается его идентификатор: `{% block title %}`, `{% block sidebar %}` или `{% block content %}`. Идентификатор разработчик придумывает сам.

- Блок **title** в HTML-теге `<title>`: здесь указано название страницы, отображаемое в заголовке окна браузера.
- Блок **sidebar** – навигация по сайту, боковое меню.
- Блок **content** – основной блок страницы, её содержание. Оно передаётся из других шаблонов. Если этот блок не переопределён, на страницу будет выведено сообщение «Контент не подвезли :(»

Тег **extends**

Одиночный тег **extends** сообщает системе, в каком шаблоне нужно переопределить блоки, описанные следом за этим тегом.

Например, если система вызывает какой-то шаблон и видит в нём тег `extends "index.html"`, то в шаблоне `index.html` будут заменены все блоки, которые в вызванном шаблоне перечислены по именам после тега `{% extends "index.html" %}`.

Создадим шаблон *list.html*, который переопределит блоки `title`, `sidebar` и `content` в шаблоне *base.html*.

```
{% extends "base.html" %}

{% block title %}Список сообщений{% endblock %}

{% block sidebar %}
    <ul>
        <li><a href="/">Главная</a></li>
        <li><a href="/about/">О сайте</a></li>
        <li><a href="/list/" class="active">Сообщения</a></li>
    </ul>
    {% endblock %}

{% block content %}
    {% for msg in messages %}
        <h2>{{ msg.title }}, от {{ msg.from }}</h2>
        <p>{{ msg.body }}</p>
    {% endfor %}
    {% endblock %}
```

Теперь из view-функции можно вызвать шаблон *list.html*, передать в него список `messages` с объектами сообщений (со свойствами `title`, `from`, `body`) — и сервер вернёт HTML-страницу с кодом из шаблона *base.html*, но с контентом, сгенерированным в файле *list.html*. То есть будет установлен `<title>` «Список сообщений», в сайдбаре будет подсвечен пункт меню «Сообщения», а в блок `content` — выведен список самих сообщений.

Пользователю будет возвращена страница *list.html*, в которой весь код будет из *base.html*, а содержимое блоков — из *list.html*.

Тег `include`

Этот тег по своей задаче — антипод тега `extend`.

Он включает в код шаблона содержимое другого шаблона. Вот простенький файл *index.html*:

```
<!DOCTYPE html>
<html>
<head>
    <title>The Last Social Media You'll Ever Need | Yatube</title>
    <link rel="stylesheet" href="style.css">
</head>

<body>
    {% include "header.html" %}
    <nav id="sidebar">
```

```

<ul>
    <li><a href="/">Главная</a></li>
    <li><a href="/about/">О сайте</a></li>
    <li><a href="/list/">Сообщения</a></li>
</ul>
</nav>

<div id="content">
    Контент страницы
</div>
{%
    include "footer.html"
%}
</body>
</html>

```

В той же директории, где сохранён **index.html**, лежат ещё два файла:

файл **header.html**

```

<div id="top-of-site">
{# здесь описана шапка сайта, она повторяется на всех страницах проекта #}
    
    <div id="site-name">Poemnotes</div>
</div>

```

файл **footer.html**

```

<div id="bottom-of-site">
{# здесь описан подвал сайта, он одинаковый на всех страницах проекта #}
     Yatube
    <div id="copyright">© Все права принадлежат всем</div>
</div>

```

В результате обработки шаблона *index.html* на место тегов `{% include "header.html" %}` и `{% include "footer.html" %}` будет вставлен код из соответствующих файлов.

В тег **include** можно передать дополнительные параметры. Например, для вывода контактов можно создать шаблон **card.html**:

```

<div class="card">
    {{ name }} <br>
    <a href="mailto:{{ email }}>Отправить сообщение</a>
</div>

```

При включении в шаблон **user.html** передаём в код шаблона *card.html* нужные данные:

```

<!DOCTYPE html>
<html>
<head>

```

```

<title>Иван Васильевич | Poemnotes</title>
<link rel="stylesheet" href="style.css">
</head>

<body>
    {% include "header.html" %}
    <nav id="sidebar">
        <ul>
            <li><a href="/">Главная</a></li>
            <li><a href="/about/">0 сайте</a></li>
            <li><a href="/list/">Сообщения</a></li>
        </ul>
    </nav>

    <div id="content">
        {% include "card.html" with name="Иван Васильевич"
email="iv@example.com" %}
        {%# а можно сделать иначе: передать значения переменных #}
        {% include "card.html" with name=user.name email=user.mail %}
    </div>
    {% include "footer.html" %}
</body>
</html>

```

13_Теги шаблонов Django: ветвления, циклы и ссылки

Ветвления

Тег ветвления (тег проверки условия) if похож на оператор ветвления в Python:

```

{% if news %}
    У вас {{ news|length }} обновлений новостей
{% elif is_holiday %}
    Новостей нет, сегодня же праздник!
{% else %}
    Сегодня нет новостей.
{% endif %}

```

В условиях {% if %} работают операторы сравнения <, >, <=, >=, !=, ==, логические операторы or, and, not, операторы тождественности is и вхождения in, круглые скобки и стандартные правила приоритета операций.

В условиях можно применять и фильтры:

```

{% if news|length >= 100 %}
    Сегодня больше сотни новостей! Что случилось?
{% endif %}

```

Циклы

Тег **for** выполняет определённый код для каждого элемента списка, переданного в цикл.

Допустим, из view-функции в шаблон передан словарь `dict_data`, в нём есть элемент `items`, в этом элементе содержится словарь, всё содержимое которого вместе с ключами нужно вывести на страницу:

```
{% for key, value in dict_data.items %}
    Ключ '{{ key }}': Значение '{{ value }}'
{% endfor %}
```

Помимо обычных переменных в циклах создаются и вспомогательные, они доступны в специальной переменной **forloop**:

- **forloop.counter** — текущий счетчик выполнений цикла, начинается с 1;
- **forloop.counter0** — текущий счетчик выполнения цикла, начинается с 0;
- **forloop.revcounter** — сколько итераций осталось до конца цикла, начинается с 1;
- **forloop.revcounter0** — сколько итераций осталось до конца цикла, начинается с 0;
- **forloop.first** — вернёт *True* на первой итерации цикла, в остальных случаях вернёт *False*;
- **forloop.last** — вернёт *True* на последней итерации цикла, в остальных случаях вернёт *False*;
- **forloop.parentloop** — если цикл был запущен внутри другого цикла, то в этой переменной находится переменная `forloop` родительского цикла.

Необязательный тег `{% empty %}`, вложенный в **for**, сработает, если переданный в цикл список пуст:

```
<ul>
{% for news in news_list %}
    <li>{{ news.title }}</li>
{% empty %}
    <li>Список новостей пуст.</li>
{% endfor %}
</ul>
```

То же самое можно написать и без `{% empty %}`, но получится громоздко:

```
<ul>
```

```
{% if news_list %}
    {% for news in news_list %}
        <li>{{ news.title }}</li>
    {% endfor %}
    {% else %}
        <li>Список новостей пуст.</li>
    {% endif %}
</ul>
```

Вложенный в **for** тег `{% ifchanged %}` запоминает значение переданных параметров или своего тела между запусками цикла, — и если они не поменялись, скрывает его.

В этом листинге HTML-заголовок `<h2>` с названием месяца будет выводиться на страницу только если в предыдущей итерации цикла название месяца было другим.

```
<h1>Архив новостей за {{ year }}</h1>

{% for news in news_items %}
    {% ifchanged %}
        <h2>{{ news.pub_date|date:"F" }}</h2>
    {% endifchanged %}

    <h3>{{ news.pub_date|date:"d.m.Y" }} | {{ news.title}}</h3>
{% endfor %}
```

В результате работы цикла получится примерно такой HTML-код:

```
<h1>Архив новостей за 2019</h1>
<h2>Январь</h2>
<h3>1.01.2019 | С Новым Годом!</h3>
<h3>2.01.2019 | С Днём научной фантастики!</h3>
<h3>3.01.2019 | С Днём рождения соломинки для коктейлей!</h3>
<h2>Февраль</h2>
<h3>1.02.2019 | С Днём работника лифтового хозяйства!</h3>
<h3>2.02.2019 | С Днём сурка!</h3>
<h3>2.02.2019 | С Днём сурка!</h3>
<h3>2.02.2019 | С Днём сурка!</h3>
```

Без тега `{% ifchanged %}` получилось бы хуже:

```
<h1>Архив новостей за 2019</h1>
<h2>Январь</h2>
<h3>1.01.2019 | С Новым Годом!</h3>
<h2>Январь</h2>
<h3>2.01.2019 | С Днём научной фантастики!</h3>
<h2>Январь</h2>
<h3>3.01.2019 | С Днём рождения соломинки для коктейлей!</h3>
<h2>Февраль</h2>
<h3>1.02.2019 | С Днём работника лифтового хозяйства!</h3>
<h2>Февраль</h2>
```

```
<h3>2.02.2019 | С Днём сурка!</h3>
<h2>Февраль</h2>
<h3>2.02.2019 | С Днём сурка!</h3>
<h2>Февраль</h2>
<h3>2.02.2019 | С Днём сурка!</h3>
```

Тег url

Тег `{% url %}` генерирует ссылки на страницы проекта. Из кода шаблона можно обратиться к именам адресов, зарегистрированных в списке **urlpatterns** в файле *urls.py*, и передать параметры, если они требуются:

```
urlpatterns = [
    path('', views.index, name='index'),
    path('/detail/<int:pk>', views.details, name='detail'),
    ...
]
```

Первый параметр тега **url** — это `name` пути из файла *urls.py*:

```
<a href="{% url 'index' %}">Главная</a>
```

После `name` при необходимости передаются значения переменных, которые принимает `path()` в *urls.py*

```
urlpatterns = [
    path('/detail/<int:pk>', views.details, name='detail'),
    path('/<str:username>/<int:id>', views.article, name='article'),
]

# передаём один параметр #
<a href="{% url 'detail' 1 %}">Подробнее об объекте 1</a> # получится ссылка
detail/1 #
<a href="{% url 'detail' pk=1 %}">Подробнее об объекте 1</a>
<a href="{% url 'detail' pk=object.id %}">Подробнее об объекте {{ object.id }}</a>

# передаём несколько параметров #
<a href="{% url 'article' 'anton' 16 %}">Статья с ID=16, автор: anton</a>
<a href="{% url 'article' username='anton' id=16 %}">Статья с ID=16, автор:
anton</a>
<a href="{% url 'article' username=article.username id=article.id %}">Статья с
ID={{ article.id }}, автор: {{ article.username }}</a>
```

14_Шаблон главной страницы

Создадим и подключим шаблон главной страницы проекта Poemnotes

Для начала подготовим директорию для хранения шаблонов. В файле настроек poemnotes/settings.py за работу шаблонов отвечает раздел TEMPLATES. Сейчас он выглядит так:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [],
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ]
        },
    },
]
```

В переменной BACKEND указано, какой язык шаблонов мы будем применять в проекте. Django поддерживает два похожих языка шаблонов: **Django Template Language (DTL)** и **Jinja2**. Оставим значение по умолчанию: наш выбор – **DjangoTemplates**

В переменной DIRS хранится список директорий, где лежат шаблоны сайта. Создайте папку templates в основной директории проекта:

```
poemnotes
├── db.sqlite3
├── manage.py
└── posts
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── migrations
    │   └── 0001_initial.py
    │       └── __init__.py
    ├── models.py
    ├── tests.py
    ├── urls.py
    └── views.py
**templates** //директория для шаблонов
└── poemnotes
    └── __init__.py
```

```
└── settings.py  
└── urls.py  
└── wsgi.py
```

Теперь в переменной `DIRS` в `poemnotes/settings.py` можно указать путь к шаблонам:

```
TEMPLATES_DIR = os.path.join(BASE_DIR, "templates")  
TEMPLATES = [  
    {  
        "BACKEND": "django.template.backends.django.DjangoTemplates",  
        "DIRS": [TEMPLATES_DIR],  
        "APP_DIRS": True,  
        "OPTIONS": {  
            "context_processors": [  
                "django.template.context_processors.debug",  
                "django.template.context_processors.request",  
                "django.contrib.auth.context_processors.auth",  
                "django.contrib.messages.context_processors.messages",  
            ]  
        },  
    },  
]
```

Шаблон главной страницы

В директории `poemnotes/templates/` создайте шаблон главной страницы `index.html` с таким кодом:

```
<!doctype html>  
<html>  
    <head>  
        <meta charset="utf-8">  
        <meta name="viewport" content="width=device-width, initial-scale=1,  
shrink-to-fit=no">  
        <title>Последние обновления | Poemnotes</title>  
    </head>  
    <body>  
        <h1>Последние обновления на сайте</h1>  
        {% for post in posts %}  
            <h3>  
                Автор: {{ post.author.get_full_name }}, дата публикации:  
{{ post.pub_date|date:"d M Y" }}  
            </h3>  
            <p>{{ post.text|linebreaksbr }}</p>  
            <hr>  
            {% endfor %}  
        </body>  
    </html>
```

Функция `render`

Теперь нужно изменить view-функцию `index` в файле `posts/views.py`. Добавьте в неё вызов шаблона `index.html` и отправку данных в этот шаблон:

```
from django.shortcuts import render
from .models import Post

def index(request):
    latest = Post.objects.order_by("-pub_date")[:11]
    return render(request, "index.html", {"posts": latest})
```

Обратите внимание, что функция `render` возвращает специальный объект, который должна вернуть view-функция. Частая ошибка — вызывать функцию, но не передать результат ее выполнения в операторе `return`.

```
# Совсем неправильно, функция вернет None: забыли return
def index_wrong(request):
    latest = Post.objects.order_by("-pub_date")[:11]
    render(request, "index.html", {"posts": latest})

# Хороший вариант: промежуточные переменные полезны
def index_ok(request):
    latest = Post.objects.order_by("-pub_date")[:11]
    response = render(request, "index.html", {"posts": latest})
    return response

# Хороший вариант: без промежуточных переменных – короче
def index_ok_too(request):
    latest = Post.objects.order_by("-pub_date")[:11]
    return render(request, "index.html", {"posts": latest})
```

Результатом будет вот такая страница:

Последние обновления на сайте

Автор: Лев Толстой, Дата публикации: 31 Jul 1854

[Фокшаны.] Еще переходъ до Фокшанъ, во время котораго я Ѳхалъ съ Монго. Человѣкъ пустой, но съ твердыми, хотя и ложными убѣжденіями. Генерал[у] по этому должно быть слушаю, угодно было спрашивать о моемъ здоровыи. Свинья! К[о]выряль нось и ничего не написаль — вотъ 2 упрека за нын[ѣшній] день. Послѣдній упрекъ становится слишкомъ часть, хотя походъ и можетъ служить въ немъ отчасти извиненiemъ. Отношенія мои съ товарищами становятся такъ пріятны, что мнѣ жалко бросить штабъ. Здоровье кажется (2) лучше.

Автор: Лев Толстой, Дата публикации: 30 Jul 1854

[Рымник.] Сдѣлалъ верхомъ переходъ до Рымника. [[5]] Стариkъ все не кланяется мнѣ. Обѣ веши эти злять меня. Съ встрѣчавшимися баши-бузуками вель себя хорошо. Объяснился съ Крыжановскимъ. Онъ, не знаю зачѣмъ, совѣтуетъ мнѣ прикомандироваться къ казачьей батареѣ; совѣтъ, которому я не послѣдую. Желчно спорилъ вечеромъ съ Фриде и Бабарыкинымъ, ругаль Сержпутовскому и ничего не сдѣлалъ, вотъ 3 упрека, которыя дѣлаю себѣ за нынѣшній день. (3)

Автор: Лев Толстой, Дата публикации: 29 Jul 1854

Исправленіе мое идетъ прекрасно. Я чувствую, какъ отношенія мои становятся пріятны и легки съ людьми всякаго рода, съ тѣхъ поръ какъ я рѣшился быть скромнымъ и убѣдился въ томъ, что казаться всегда величественнымъ и непогрѣшнымъ вовсе не есть необходимость. Я очень весель. И дай Богъ, какъ мнѣ кажется, — чтобы веселье это происходило отъ самаго меня, отъ желаній всѣхъ. Быть пріятнымъ, скромности, необходимости и внимательности за

15_Базовый шаблон

В прошлом уроке мы сделали шаблон главной страницы. Это отличный промежуточный результат работы над проектом, но случайный пользователь останется недоволен: проекту не хватает оформления. Wiki может себе такое позволить, а мы пока нет.

Добавим в проект поддержку CSS и JavaScript.

С **HTML** вы немного знакомы — это язык разметки веб-страницы, её скелет.

CSS, таблица стилей, отвечает за оформление страницы. Цвет, размер и начертание шрифтов, обрамление и фон элементов страницы, расстояния между элементами и блоками текста, фоновые изображения — всё это описывается в стилевых файлах, связанных с файлом HTML.

JavaScript управляет динамическим поведением страницы, от интерактивных блоков до подгрузки нового содержимого на страницу. Код JavaScript пишется в файлах `*.js`, привязанных к HTML-файлу.

Добавление директории для статических файлов

Статическими файлами, «статикой», называют файлы, которые не модифицируются сервером в момент обращения. Эти файлы скачиваются браузером «как есть».

В Django каждое приложение может иметь свой набор статических файлов. Это могут быть картинки, стили, JS или другие файлы, необходимые для работы конкретного приложения. Эти файлы хранятся в папке **static** в приложении.

Перед публикацией проекта на боевом сервере все статические файлы проекта собираются в единую директорию. Сборка запускается командой `collectstatic`.

По этой команде Django проверит все директории проекта и скопирует в общую директорию содержимое всех папок с названием `static`, какие найдёт в проекте. Если по каким-то причинам необходимо назвать папку со статикой иным именем — адрес такой папки нужно зарегистрировать в переменной `STATICFILES_DIRS` в `poemnotes/settings.py`

Адрес общей директории для статики указывается в настройках проекта.

В реальных проектах за отдачу статических файлов часто отвечают отдельные сервера или CDN-сервисы (англ. "content delivery network", «сеть доставки контента»), единственная задача которых — заниматься раздачей статических файлов. После сборки статические файлы из общей папки будет удобнее загрузить на сервер, раздающий статику вашего проекта.

Создайте две директории с названием `static`: одну в корне проекта, для сборки, вторую — в приложении `posts`, для статики приложения. Получится такая структура:

```
poemnotes
├── db.sqlite3
├── manage.py
└── posts
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── migrations
    │   ├── 0001_initial.py
    │   └── __init__.py
    ├── models.py
    ├── static # директория для статических файлов приложения posts
    ├── tests.py
    ├── urls.py
    └── views.py
├── static # директория для сборки статических файлов проекта
└── templates
    └── index.html
└── poemnotes
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

В файл poemnotes/settings.py после переменной STATIC_URL добавьте новую переменную STATIC_ROOT:

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/2.2/howto/static-files/

# задаём произвольный URL, который будет использоваться для запросов к
# статическим файлам
STATIC_URL = "/static/"
# теперь логотип можно будет запросить по адресу sitename.ex**/static/
**images/logo.png

# задаём адрес директории, куда командой *collectstatic* будет собрана вся
# статика
STATIC_ROOT = os.path.join(BASE_DIR, "static")
```

При разработке не обязательно выполнять команду collectstatic. Командой runserver запустите встроенный веб-сервер Django — и он будет отдавать статику прямо из исходных директорий.

Шаблон на Bootstrap

Вам не понадобится самостоятельно настраивать дизайн проекта, за основу возьмём фреймворк [Bootstrap](#) версии 4.3

Этот проект произвёл небольшую революцию в мире разработки сайтов. Он дал возможность backend-программистам быстро получать приемлемый дизайн проектов, не погружаясь в детали фронтенд-разработки. Взял, поставил — готово.

Мы подготовили архив с необходимой статикой. Скачайте и распакуйте его в posts/static.

<https://code.s3.yandex.net/backend-developer/learning-materials/static.zip>

Включение статических файлов в шаблоны

После того как вы собрали статику и запустили проект с помощью команды runserver, можете попробовать открыть какой-нибудь файл из директории STATIC_ROOT через HTTP.

Например, по ссылке <http://127.0.0.1:8000/static/bootstrap/dist/js/bootstrap.js> в браузере откроется код JavaScript-библиотеки.

Ссылка состоит из нескольких частей:

- http://127.0.0.1:8000/ — адрес вашей локальной машины и порт, на котором запущен Django.
- /static/ — эта часть пути управляется служебным приложением Django `django.contrib.staticfiles`. Поменять эту часть пути вы можете в переменной `STATIC_URL` в файле настроек проекта `yatube/settings.py`
- bootstrap/dist/js/bootstrap.js — это путь к файлу в папке `STATIC_ROOT`

Чтобы добавить ссылку на подгрузку файла в шаблон, необходимо загрузить модуль для работы со статикой командой `{% load static %}`, а в адресах подключаемых файлов применять тег `{% static "адрес_файла_относительно_директории_статики" %}`:

```
{% load static %}  
<script src="{% static "bootstrap/dist/js/bootstrap.js" %}"></script>
```

Конструкция кажется громоздкой, но на практике это сильно экономит время. Если нужно разместить статику отдельно от проекта, на CDN-сервере — потребуется лишь изменить несколько настроек конфигурации, и проект заработает и на локальном компьютере разработчика, и на боевых серверах.

base.html

Создадим базовый шаблон. Он будет содержать обрамление, повторяющееся на всех страницах. В HTML-коде это будет невидимый пользователю служебный блок `<head>` с подключенными css и js-файлами.

Уникальные части шаблонов будем хранить в отдельных файлах.

Создадим базовый HTML-файл на основе [примера](#) из документации по Bootstrap и определим стандартные блоки для заголовка страницы и тела:

```
<!doctype html>  
<html>  
  
<head>  
    <meta charset="utf-8">  
    <meta name="viewport" content="width=device-width, initial-scale=1,  
    shrink-to-fit=no">  
    <title>{% block title %}The Last Social Media You'll Ever Need{% endblock  
%} | Poemnotes</title>  
    <!-- Загрузка статики -->
```

```

    {% load static %}
    <link rel="stylesheet" href="{% static 'bootstrap/dist/css/bootstrap.min.css' %}">
    <script src="{% static 'jquery/dist/jquery.min.js' %}"></script>
    <script src="{% static 'bootstrap/dist/js/bootstrap.min.js' %}"></script>
</head>

<body>

    <main>
        <div class="container">
            <h1>{% block header %}The Last Social Media You'll Ever Need{% endblock %}</h1>
            {% block content %}
                <!-- Содержимое страницы -->
            {% endblock %}
        </div>
    </main>

</body>
</html>

```

Сохраните этот файл в папку `templates` под именем `base.html`. Теперь измените шаблон главной страницы `index.html`:

```

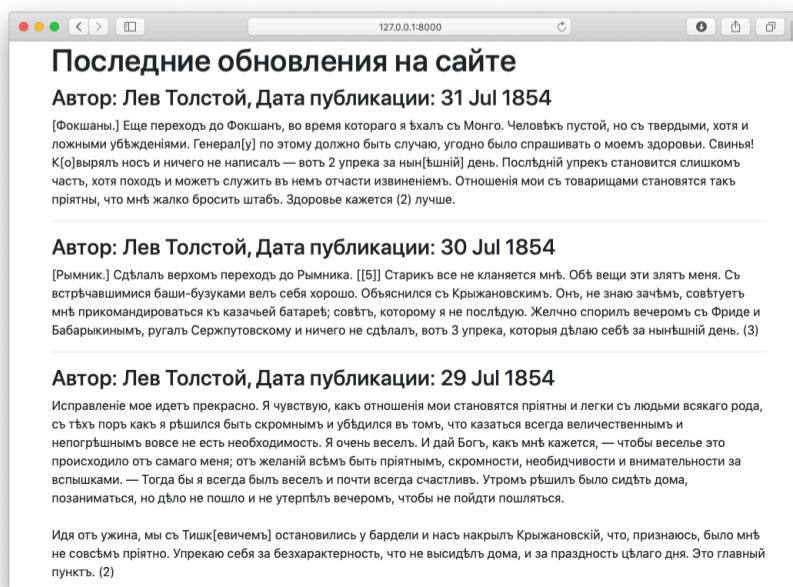
{% extends "base.html" %}
{% block title %}Последние обновления на сайте{% endblock %}
{% block header %}Последние обновления на сайте{% endblock %}
{% block content %}

    {% for post in posts %}
        <h3>
            Автор: {{ post.author.get_full_name }}, Дата публикации:
        {{ post.pub_date|date:"d M Y" }}
        </h3>
        <p>{{ post.text|linebreaksbr }}</p>
        {% if not forloop.last %}<hr>{% endif %}
        {% endfor %}

    {% endblock %}

```

Обновите главную страницу сайта, после всех действий она должна выглядеть более нарядно:



Теперь на основе базового шаблона мы сможем создавать новые страницы, сохраняя общую структуру и стилистику. А для изменения общей структуры потребуется изменить лишь один шаблон.

16 CRUD и фильтрация через ORM

Для работы с БД у моделей в Django есть встроенный набор методов. Они наследуются от класса `models.Model` и поддерживают основные операции по обработке данных в БД: **CRUD**. Вы знакомы с этой аббревиатурой из урока по SQL

CRUD-операции

- **Create:** `Model.objects.create()` — создание объекта в базе
- **Read:** `Model.objects.get(id=N)` — чтение объекта по его ключу
- **Update:** `object.property= 'new value'` и потом `object.save()` — изменение объекта
- **Delete:** `object.delete()` — удаление объекта из базы

Сейчас мы разберёмся с основными задачами, которые решает Django ORM. Но перед этим познакомимся с инструментом, который упростит нам тестирование кода.

Python shell

С интерпретатором кода Python, как и со многими другими программами, можно работать через командную строку. Если в консоли выполнить команду `$python3` без параметров, то интерпретатор Python запустится в «интерактивном режиме». Теперь можно ввести в командную строку любые скрипты `python` — и они будут выполняться прямо в терминале. Это похоже на работу командной строки, но вместо команд для работы с файлами выполняется программный код, строчка за строчкой.

Откройте новое окно терминала и посмотрите, как работает `python shell`.

Символы `>>>` — это приглашение для ввода команд, то же, что и знак `$` в командной строке.

```
# запускаем интерпретатор без параметров
$ python3
```

```
# дальше пишем на python
# создаём переменную
>>> best_slogan = "Mischief Managed!"
# вызываем функцию print()
>>> print(best_slogan)
# и получаем результат
Mischief Managed!
# арифметика тоже работает
>>> 2 + 2 * 2
6
```

Прелесть этого режима в том, что он тут же выводит результат выполнения скрипта. Например, создадим и выведем переменную:

```
>>> x = 5
>>> x
5
```

Обратите внимание: переменные, которые вы создали во время работы в `python shell`, будут доступны до тех пор, пока вы не закроете окно терминала.

Django Python shell

В таком же интерактивном режиме можно работать и с Django-проектами. Для этого `python shell` надо запустить в виртуальном окружении проекта.

Откройте терминал, убедитесь, что запущено виртуальное окружение проекта Poemnotes и выполните команду:

```
(venv) $ python manage.py shell
```

Вы увидите примерно такой результат:

```
(venv) $ python manage.py shell
Python 3.8.0 (default, Nov 22 2019, 23:37:58)
[Clang 11.0.0 (clang-1100.0.33.12)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

Всё, что вы хотите узнать, но стесняетесь спросить — выясняйте через команду `help`.

При работе в интерактивном режиме в Django вам становятся доступны все данные проекта. Можно создавать объекты, управлять базой данных, тестировать функции проекта.

```
(venv) $ python manage.py shell
```

```
Python 3.8.0 (default, Nov 22 2019, 23:37:58)
[Clang 11.0.0 (clang-1100.0.33.12)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
# чтобы убедиться, что мы работаем именно с нашим проектом Yatube
# импортируем модели из проекта и заглянем в базу данных:
# запросим все объекты модели Post
>>> from posts.models import Post, User
>>> Post.objects.all()
```

Основы работы с shell разобрали, теперь можно и делами заняться. Все следующие команды выполняйте в shell.

Работаем с проектом

Запрос к базе возвращает специальный объект **QuerySet**, который содержит список объектов, соответствующих условиям запроса. По запросу `.all()` мы получили все объекты модели `Post`.

Чтобы получить определённый объект, можно обратиться к нему по его *primary key*:

```
# можно использовать User.objects.get(id=1)
>>> me = User.objects.get(pk=3)
>>> me
# python shell сообщает, что переменная me содержит <Объект> класса User,
# а поле username этого объекта равно "admin"
<User: admin>
```

Класс `User` предустановлен в Django, и для него настроен вывод на экран именно в таком виде. При создании любого класса можно описать, каким образом объекты этого класса будут выводиться на экран (например, в `python shell` или при вызове `print(any_object)`). Это описывается в «магическом методе» `__str__`.

Запросим пользователя с `pk=13`: у нас в базе такой записи пока что нет. Если объект с запрошенным ключом не найден, то появится сообщение об ошибке:

```
>>> User.objects.get(pk=13)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
    File "/Dev/Poemnotes/venv/lib/python3.8/site-packages/django/db/models/
manager.py", line 82, in manager_method
        return getattr(self.get_queryset(), name)(*args, **kwargs)
    File "/Dev/Poemnotes/venv/lib/python3.8/site-packages/django/db/models/
query.py", line 415, in get
        raise self.model.DoesNotExist
```

```
django.contrib.auth.models.User.DoesNotExist: User matching query does not exist.
```

Новый объект в базе можно создать методом **create()**:

```
# создаём объект, передаём свойства
>>> new = Post.objects.create(author=me, text="Смотри, этот пост я создал
через shell!")
# посмотрим, какой id присвоен этому объекту в базе
>>> new.id
39
# а что в поле text?
>>> new.text
"Смотри, этот пост создан через shell!"
# а что в поле author?
>>> new.author
<User: admin>
# смотрим, что записано в поле username того объекта, на который ссылается
поле author
>>> new.author.username
'admin'
```

Объект `new` в момент создания сохранился в базе и получил уникальный *id*.

Чтобы изменить этот объект, надо присвоить новое значение одному из его полей и вызвать метод **save()**:

```
# присваиваем новое значение полю text
>>> new.text = "Смотри, этот пост обновлён!"
# но пока что значение изменено лишь в коде. В БД всё ещё хранится старое
значение
# чтобы отправить новое значение в базу данных – вызываем метод save()
>>> new.save()
```

Если вы изменили объект в коде – он не изменится в базе до тех пор, пока вы не вызовете метод `save()`.

Теперь обновлённый текст записи можно увидеть в админ-зоне.

Удалить объект из базы можно методом **delete()**. При вызове этот метод дополнительно удалит и все связанные объекты, для которых был задан параметр `on_delete=models.CASCADE`.

```
>>> new.delete()
```

Для дальнейшей работы нам понадобятся тестовые посты админа, создайте их:

```
>>> first_post = Post.objects.create(author=me, text="Oops, I did it again!")
>>> second_post = Post.objects.create(author=me, text="Утромъ гольдъ Дерсу
Узала на повторно заданный вопросъ согласенъ ли онъ поступить проводникомъ
изъявилъ свое согласie и съ этого момента онъ сталъ членом экспедиції")
```

Фильтрация объектов

Основная задача при работе с базой — это поиск объектов по заданным признакам. В SQL за это отвечают команды блока WHERE, в Django ORM — метод `filter()`:

```
# найти все объекты, значение поля author у которых равно me
# в этой переменной хранится объект User с pk=3
>>> Post.objects.filter(author=me)
<QuerySet [<Post: Oops, I did it again!>, <Post: Утромъ гольдъ Дерсу Узала...>]>
```

В базе найдены две записи, соответствующие условиям запроса.

Если ваш вывод выглядит так `<QuerySet [<Post: Post object(39)>, <Post: Post object(40)>]` значит в классе `Post` не хватает метода `__str__`, который нужен для красивого вывода объектов на экран. Добавьте его:

```
class Post:
    ...
    def __str__(self):
        # выводим текст поста
        return self.text
```

Увидеть SQL-запрос, который будет отправлен к базе, можно с помощью команды `.query`:

```
>>> print(Post.objects.filter(author=me).query)
SELECT "posts_post"."id", "posts_post"."text", "posts_post"."pub_date",
"posts_post"."author_id", "posts_post"."group_id" FROM "posts_post" WHERE
"posts_post"."author_id" = 1
```

В Django ORM аналог команды WHERE выглядит так: указывается имя поля, затем два знака подчеркивания `__`, название фильтра и его значение:

```
# найти посты, где поле text__contains содержит строку "again"
>>> Post.objects.filter(text__contains='again')
<QuerySet [<Post: Oops, I did it again!>]>
```

При запросе указываются именованные параметры функции `filter()`. Имя параметра состоит из имени поля и суффикса, указывающего, какой оператор применять. Доступные операторы:

- **exact** — точное совпадение. «Найти пост, где поле `id` точно равно 1»

ORM: `Post.objects.filter(id__exact=1)` или `Post.objects.filter(id=1)`

На SQL это условие выглядит так: `SELECT ... WHERE id = 1.`

Сравнение работает и с `None`.

Выражение `Post.objects.filter(text=None)` превратится в `SELECT ... WHERE text IS NULL`

- **contains** – поиск по тексту в поле `text`. «Найти пост, где в поле `text` есть слово "oops" именно в таком регистре»

ORM: `Post.objects.filter(text__contains="oops")`

SQL: `SELECT ... WHERE text LIKE '%oops%';`

В большинстве баз данных (например, в MySQL или PostgreSQL) ничего не найдётся: регистр символов не совпадает. В посте админа написано "*Oops*", а в запросе — "oops".

Однако в нашем проекте установлена СУБД **SQLite** (Django ставит её по умолчанию), и у неё есть неприятная особенность: она не различает регистр символов нигде, кроме как в кодировке **ASCII** (а мы все давно уже работаем в **UTF-8**, ведь в ней есть смайлики 😊).

Мануалы формулируют проблему так: *SQLite only understands upper/lower case for ASCII characters by default.*

Итак: в

базе **SQLite** фильтр `Post.objects.filter(text__contains="oops")` найдёт записи со словами *oops*, *Oops* и *OOPS*, а в большинстве других баз такой запрос вернёт только запись, где регистр слова совпадает полностью.

- **in** – вхождение в множество. «Найти пост, где значение поля `id` точно равно одному из значений: 1, 3 или 4»

ORM: `Post.objects.filter(id__in=[1, 3, 4])`

SQL: `SELECT ... WHERE id IN (1, 3, 4);`

Если вместо списка будет передана строка, она разобьётся на символы:
«Найти пост, где значение поля *text* точно равно "o", "p" или "s"»

ORM: `Post.objects.filter(id__in="oops")`

SQL: `SELECT ... WHERE text IN ('o', 'p', 's');`

- Операторы сравнения

gt — `>` (больше),

gte — `=>` (больше или равно),

lt — `<` (меньше),

lte — `<=` (меньше или равно).

«Найти пост, где значение поля *id* больше пяти»

ORM: `Post.objects.filter(id__gt=5)`

SQL: `SELECT ... WHERE id>5;`

- Операторы сравнения с началом и концом строки **startswith**, **endswith**

«Найти посты, где содержимое поля *text* начинается со строки "Утромъ"»

ORM: `Post.objects.filter(text__startswith="Утромъ")`

SQL: `SELECT ... WHERE text LIKE Утромъ% ESCAPE`

- **range** — вхождение в диапазон

```
import datetime
start_date = datetime.date(1890, 1, 1)
end_date = datetime.date(1895, 3, 31)
Post.objects.filter(pub_date__range=(start_date, end_date))
# SQL: SELECT ... WHERE pub_date BETWEEN '1890-01-01' and '1895-03-31';
# выберет посты, опубликованные в диапазоне с 1 января 1890 до 31 марта 1895
```

- При работе с частями дат можно применять дополнительные суффиксы **date**, **year**, **month**, **day**, **week**, **week_day**, **quarter** и указывать для них дополнительные условия:

```
# условия для конкретной даты
Post.objects.filter(pub_date__date=datetime.date(1890, 1, 1))
Post.objects.filter(pub_date__date__lt=datetime.date(1895, 1, 1))
# условия для года и месяца
Post.objects.filter(pub_date__year=1890)
Post.objects.filter(pub_date__month__gte=6)
# условия для квартала
Post.objects.filter(pub_date__quarter=1)
```

Такой же синтаксис применяется и для времени: **hour**, **minute**, **second**.

- **isnull** — проверка на пустое значение.

ORM: `Post.objects.filter(pub_date__isnull=True)`

SQL: `SELECT ... WHERE pub_date IS NULL;`

Объединение условий

В одном запросе можно указать несколько условий одновременно. Для этого последовательно вызовите методы `filter()` с различными параметрами. Будет сгенерирован SQL-запрос, в котором все условия объединены оператором `AND`.

Исключить данные из выборки можно методом `exclude()`:

```
# выбрать посты, начинающиеся со слова "Утромъ"
# исключить из выборки посты автора me
# и показать только те посты, которые опубликованы не ранее 30 января 1895
года
>>> Post.objects.filter(
...     text__startswith='Утромъ'
... ).exclude(
...     author=me
... ).filter(
...     pub_date__gte=datetime.date(1895, 1, 30)
... )
```

Сортировка и ограничение количества результатов

Этот синтаксис вам знаком: мы применяли сортировку во view-функции `index()` и там же ограничили число возвращаемых результатов запроса.

`order_by("-pub_date")` — сортировать результаты по полю `pub_date` в обратном порядке (от больших значений к меньшим) `[:11]` — вернуть не более одиннадцати результатов из найденных.

```
>>> print(Post.objects.order_by("-pub_date")[:11].query)
SELECT "posts_post"."id", "posts_post"."text", "posts_post"."pub_date",
"posts_post"."author_id", "posts_post"."group_id" FROM "posts_post" ORDER
BY "posts_post"."pub_date" DESC LIMIT 11
```

Сортировку и ограничение числа возвращаемых результатов можно объединить с фильтрацией:

```
>>> Post.objects.filter(text__startswith='Утромъ').order_by("-pub_date")[:2]
```

В Django ORM есть и дополнительный синтаксис, он описан в [документации](#).

17_Дополнительные возможности ORM

При знакомстве с SQL вы строили запросы через объединение таблиц `JOIN` и применяли агрегирующие функции `MAX`, `MIN`, `COUNT` etc. Посмотрим, как эти инструменты работают в Django ORM.

Вывод запросов в консоли

Для более наглядного понимания того, как Django ORM работает с базой данных, активируем вывод отладочной информации. В виртуальном окружении запустите `python shell`:

```
(venv) $ python manage.py shell
```

Следующая команда импортирует библиотеку `logging`, входящую в стандартный набор модулей, и включит вывод отладочной информации для той части Django, которая делает запросы к базе:

```
>>> import logging
```

```
>>> log = logging.getLogger('django.db.backends')
>>> log.setLevel(logging.DEBUG)
>>> log.addHandler(logging.StreamHandler())
```

Этот режим останется активным до конца работы в консоли. По ссылке вы можете посмотреть [документацию о работе библиотеки logging](#).

Теперь в консоли будут отображаться SQL-запросы, сформированные Django ORM при обращении к базе.

Импортируем модели и создадим переменную `latest`:

```
>>> from posts.models import Post, User
>>> latest = Post.objects.all()
```

Однако SQL-запрос не отобразился в консоли! Дело в том, что мы просто создали код запроса, но никаких данных не запросили. Это достаточно разумная оптимизация: Django старается лишний раз не дёргать базу.

Запустим в терминале код шаблона, который выводит на главную страницу свежие записи. Чтобы не загромождать вывод текстами записей — сохраним их во временной строке `tmp`, но не будем выводить эту строку на экран:

```
>>> for post in latest:
...     tmp = f'{post.text} Автор {post.author.username}'
...
(0.000) SELECT "posts_post"."id", "posts_post"."text",
"posts_post"."pub_date", "posts_post"."author_id", "posts_post"."group_id"
FROM "posts_post"; args=()
(0.000) SELECT "auth_user"."id", "auth_user"."password",
"auth_user"."last_login", "auth_user"."is_superuser", "auth_user"."username",
"auth_user"."first_name", "auth_user"."last_name", "auth_user"."email",
"auth_user"."is_staff", "auth_user"."is_active", "auth_user"."date_joined"
FROM "auth_user" WHERE "auth_user"."id" = 2 LIMIT 21; args=(2,)
[...skip...]
```

Разберёмся с этим выводом.

Мы запросили данные для модели **Post**. Её свойство `author` — ссылка на модель **User**, в `author` хранится первичный ключ записи из таблицы **User**.

При попытке обработать данные Django поступил как ленивый сотрудник: ему сказали принести из базы данных содержимое таблицы `posts`, он сходил и принёс; ему сказали принести из модели **User** свойство `username`, он опять сходил и принёс. За каждой записью он ходил по очереди.

Наверное, чтобы приготовить омлет, он ходит в магазин за каждым яйцом отдельно.

А ведь связей может быть несколько: в нашем проекте есть модель для сообществ, и если будет нужно показать принадлежность поста к сообществу, мы получим ещё один запрос. В результате ресурсоёмкость проекта может увеличиться многократно: для отображения одной страницы потребуются сотни SQL-запросов!

Поочерёдное получение записей из базы требует гораздо больших ресурсов, чем единый запрос. Нам срочно нужен `JOIN` для запросов через Django ORM!

Загрузка связанных записей

Django предлагает два способа загрузки связанных записей:

- `select_related(relation)` — загрузка связанных данных с помощью `JOIN`. В результате обработки получается один запрос, который, помимо основной модели, загружает и связанные данные из дополнительных таблиц.
- `prefetch_related(relation)` — «ленивая» подгрузка связанных данных с помощью дополнительных запросов. В этом случае Django ORM сперва запрашивает данные из основной таблицы, запоминает первичные ключи связанных записей, а затем делает ещё один запрос для загрузки связанных данных, ключи которых есть в первой выборке.

Выбор подходящего варианта зависит от характеристики данных и ситуации.

Если данных мало, то один запрос создаст меньшую нагрузку на базу и на Python, который потом эти данные будет обрабатывать. Выбор в пользу `select_related()`.

А в ситуации, когда в таблице связанной модели лежат огромные объекты (например, файлы с картинками или большие тексты), можно оптимизировать нагрузку, не пересыпая одни и те же данные много раз.

Представьте, что в модели **User**, помимо *username* и прочей служебной информации, хранятся портреты пользователей, большие картинки. Мы запросили список постов, и при получении каждого поста мы получаем из связанной таблицы информацию об авторе этого поста. Для тридцати постов информация об авторах будет передана тридцать раз. С картинками.

Но в базе у нас только два автора, и каждый из постов написан одним из них! Значит, одну и ту же информацию мы запрашиваем многократно, впустую расходуя ресурсы сервера, увеличивая трафик и время обработки данных.

Избежать таких расходов поможет запрос `prefetch_related()`.

Посмотрим, как это выглядит в коде.

Методам `select_related()` и `prefetch_related()` параметром передаём имя поля, в котором хранятся ключи связанной модели.

Сравните запросы:

```
>>> related = Post.objects.select_related('author').all()
>>> for post in related:
...     tmp = f'{post.text} Автор {post.author.username}'
...
# запрашиваем данные FROM "posts_post"
# и, дополнительно, данные автора INNER JOIN "auth_user":
# всё в одном запросе
(0.000) SELECT "posts_post"."id", "posts_post"."text",
"posts_post"."pub_date", "posts_post"."author_id", "posts_post"."group_id",
"auth_user"."id", "auth_user"."password", "auth_user"."last_login",
"auth_user"."is_superuser", "auth_user"."username", "auth_user"."first_name",
"auth_user"."last_name", "auth_user"."email", "auth_user"."is_staff",
"auth_user"."is_active", "auth_user"."date_joined" FROM "posts_post" INNER
JOIN "auth_user" ON ("posts_post"."author_id" = "auth_user"."id"); args=()
>>>
```

Для `select_related` хватило только одного запроса!

А вот так работает `prefetch_related`:

```
>>> related = Post.objects.prefetch_related('author').all()
>>> for post in related:
...     tmp = f'{post.text} Автор {post.author.username}'
...
# запрашиваем все посты FROM "posts_post"
```

```
(0.000) SELECT "posts_post"."id", "posts_post"."text",
"posts_post"."pub_date", "posts_post"."author_id", "posts_post"."group_id"
FROM "posts_post"; args=()
# а теперь запрашиваем авторов FROM "auth_user", но только с перечисленными
id:
# WHERE "auth_user"."id" IN (1, 2)
# Django ORM получил список необходимых id из результатов первого запроса
(0.000) SELECT "auth_user"."id", "auth_user"."password",
"auth_user"."last_login", "auth_user"."is_superuser", "auth_user"."username",
"auth_user"."first_name", "auth_user"."last_name", "auth_user"."email",
"auth_user"."is_staff", "auth_user"."is_active", "auth_user"."date_joined"
FROM "auth_user" WHERE "auth_user"."id" IN (1, 2); args=(1, 2)
```

В этот раз запросов два. Сначала Django ORM запросил посты, а потом — информацию об авторах, но только о тех, которые упомянуты в результирующей выборке первого запроса.

Вызовите цикл ещё раз и обратите внимание на то, сколько запросов будет отправлено к базе при повторном использовании переменных в зависимости от выбранной стратегии.

18_Агрегирующие функции в Django ORM

С агрегирующими функциями в SQL вы знакомы: они возвращают из БД сумму, максимум или среднее арифметическое по определённому полю для всех записей в таблице или для выборки: «SQL, выбери в таблице *joke* все записи за 1.04.2001 и примени к ним агрегирующую функцию: найди запись с максимальным значением в столбце *likes*».

База данных подсчитает и выведет полученные данные в отдельный столбец результирующей выборки.

Для группировки строк используется специальный запрос GROUP BY, в котором перечисляются столбцы, по которым должна идти группировка. Это значит, что если в таблице несколько строк с одним и тем же значением в указанном поле, то такие строки объединяются в одну, а над другими столбцами можно провести групповые операции.

Слоноводы несколько дней кормят слонов ивовыми вениками и записывают, кто сколько съел.

| ID | DATE | NAME | BROOM |
|----|------------|-----------|-------|
| 1 | 2019-08-17 | Аристарх | 43 |
| 2 | 2019-08-17 | Джульетта | 38 |
| 3 | 2019-08-17 | Кузя | 26 |
| 4 | 2019-08-18 | Аристарх | 35 |
| 5 | 2019-08-18 | Джульетта | 33 |
| 6 | 2019-08-18 | Кузя | 19 |
| 7 | 2019-08-19 | Аристарх | 35 |

Затем слоноводы хотят найти для каждого слона максимальное количество съеденного за один день, и для этого делают запрос к базе.

```
SELECT date, name, MAX(broom)
FROM brooms
GROUP BY name;
```

«Под капотом» происходит следующее: сначала в колонке **name** будут найдены и сгруппированы все совпадающие записи, получится три группы: все записи по Аристарху:

| ID | DATE | NAME | BROOM |
|----|------------|----------|-------|
| 1 | 2019-08-17 | Аристарх | 43 |
| 4 | 2019-08-18 | Аристарх | 35 |
| 7 | 2019-08-19 | Аристарх | 35 |
| 10 | 2019-08-20 | Аристарх | 34 |
| 13 | 2019-08-21 | Аристарх | 44 |

все записи по Джульетте:

| ID | DATE | NAME | BROOM |
|----|------------|-----------|-------|
| 2 | 2019-08-17 | Джульетта | 38 |
| 5 | 2019-08-18 | Джульетта | 33 |
| 8 | 2019-08-19 | Джульетта | 41 |
| 11 | 2019-08-20 | Джульетта | 42 |
| 14 | 2019-08-21 | Джульетта | 39 |

и все записи по Кузе:

| ID | DATE | NAME | BROOM |
|----|------------|------|-------|
| 3 | 2019-08-17 | Кузя | 26 |
| 6 | 2019-08-18 | Кузя | 19 |
| 9 | 2019-08-19 | Кузя | 9 |
| 12 | 2019-08-20 | Кузя | 18 |
| 15 | 2019-08-21 | Кузя | 23 |

Затем функция MAX(broom) в каждой из этих групп найдет максимальное значение в колонке broom, и в результирующую выборку будут выведены имена и найденные значения.

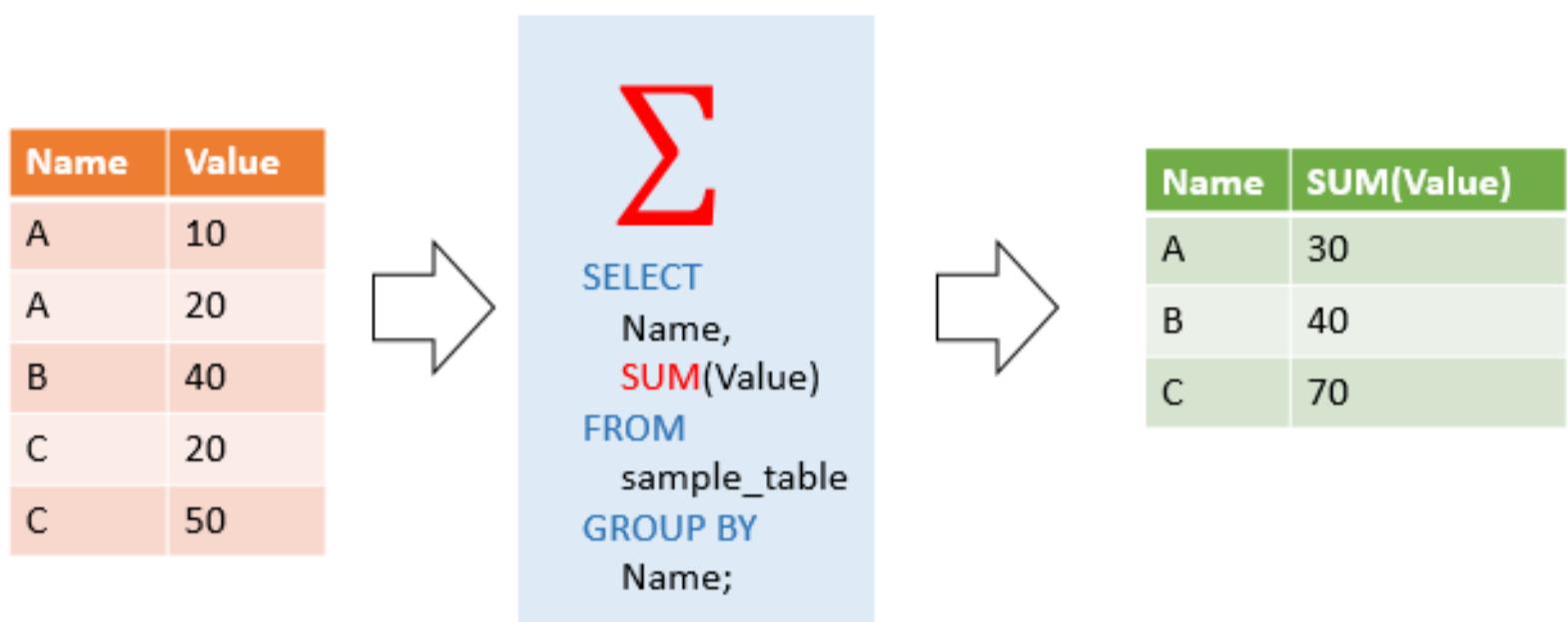
Результат запроса будет таким:

| DATE | NAME | MAX(BROOM) |
|------------|-----------|------------|
| 2019-08-21 | Аристарх | 44 |
| 2019-08-20 | Джульетта | 42 |
| 2019-08-17 | Кузя | 26 |

Если в БД какого-нибудь магазина в таблице хранятся чеки покупок, можно сгруппировать записи по **id** покупателя, после чего применить агрегирующую функцию к полю, где указана цена чека — и посчитать максимальную или среднюю сумму заказов определённого покупателя.

А можно просто сложить все **value** по отдельности для A, B и C:

Если не указать **GROUP BY** — агрегирующая функция будет применена ко всей таблице. Например `SELECT SUM(Value) FROM sample_table;` вернет значение 140 для примера на иллюстрации, а `SELECT MAX(broom) FROM brooms;` вернёт 44, максимальное значение во всей колонке broom из таблицы со слонами и вениками.



Метод count()

Задача подсчёта записей в таблице или в выборке столь популярна, что в Django ORM она получила несколько вариантов решения. Чтобы узнать количество полученных строк, можно вызывать **метод count()** для `objects`, дописав его к конструкции, делающей выборку:

```

# выбираем посты, опубликованные позже (gt) июня 1854 года, затем
# пересчитываем их
>>> Post.objects.filter(pub_date__month__gt=6, pub_date__year=1854).count()

(0.012) SELECT COUNT(*) AS "__count" FROM "posts_post" WHERE
(django_datetime_extract('month', "posts_post"."pub_date", 'UTC') > 6 AND
"posts_post"."pub_date" BETWEEN '1854-01-01
00:00:00' AND '1854-12-31 23:59:59.999999'); args=(6, '1854-01-01 00:00:00',
'1854-12-31 23:59:59.999999')
30
  
```

Когда нужно узнать лишь число записей, но сами записи не нужны — не вызывайте метод `all()`, применяйте `count()`.

Метод `all()` заставит базу прочитать и передать в код весь объём данных, а `count()` выполнит всю работу на стороне базы и вернет лишь одно число. Загрузка и обработка всех данных в такой ситуации — это пустой расход ресурсов.

```

# Проверка: есть ли данные в таблице

# Неправильный способ: этот код загружает вообще все данные из таблицы
if User.objects.all():
    print("Пользователи есть")

# Правильный способ: этот код просит базу вернуть число
  
```

```
if User.objects.count():
    print("Пользователи есть")
```

Метод aggregate()

Метод `aggregate()` применяет **агрегирующие функции** к определённой выборке или ко всей таблице.

В Django есть несколько агрегирующих функций, вот самые популярные из них:

- **Avg**: вернёт среднее значение по указанной колонке в выборке
- **Count**: вернёт количество записей в выборке, как и метод `count()`, описанный выше
- **Max**: вернёт максимальное значение по указанной колонке в выборке
- **Min**: вернёт минимальное значение по указанной колонке в выборке
- **Sum**: вернёт сумму значений по указанной колонке в выборке

Эти функции хранятся в модуле `django.db.models`, перед применением их надо импортировать в код.

Найдём самый большой *id* и пересчитаем объекты в модели *Post* при помощи метода `aggregate()`:

```
>>> from django.db.models import Max, Count
# найти максимальное значение id для объектов Post
>>> Post.objects.aggregate(Max("id"))
(0.000) SELECT MAX("posts_post"."id") AS "id__max" FROM "posts_post"; args=()
{'id__max': 43}

# пересчитать объекты id в модели Post
>>> Post.objects.aggregate(Count("id"))
(0.000) SELECT COUNT("posts_post"."id") AS "id__count" FROM "posts_post";
args=()
{'id__count': 37}
37
```

Связи между таблицами

При создании модели **Post** мы добавили в неё ссылку на автора, на модель **User**, и указали `related_name="posts"`.

```
class Post(models.Model):
    # ... какой-то код
    author = models.ForeignKey(
        User, on_delete=models.CASCADE, related_name="posts"
    )
```

У модели `User` автоматически появится свойство `posts`, оно ссылается на все записи текущего автора.

```
>>> leo = User.objects.get(id=2)
(0.000) SELECT "auth_user"."id", "auth_user"."password",
"auth_user"."last_login", "auth_user"."is_superuser", "auth_user"."username",
"auth_user"."first_name", "auth_user"."last_name", "auth_user"."email",
"auth_user"."is_staff", "auth_user"."is_active", "auth_user"."date_joined"
FROM "auth_user" WHERE "auth_user"."id" = 2; args=(2,)
>>> leo.username
'leo'

>>> leo.posts.count()
# leo.posts – это выборка тех объектов из модели Post,
# у которых в поле author_id стоит "2" (которые связаны с leo),
# потому что leo – это объект User с id=2
(0.000) SELECT COUNT(*) AS "__count" FROM "posts_post" WHERE
"posts_post"."author_id" = 2; args=(2,)
```

36

Свойство `posts` у объекта модели `User` появилось в результате того, что модель `Post` ссылается на `User`. И, благодаря магии ORM, мы можем получить все записи автора, обратившись к свойству `posts`.

Как только вы создаёте в модели поле со ссылкой на другую модель — у второй модели появляется новое свойство, «обратная связь», и к этому свойству можно обращаться, как к любому другому.

Свойства, ссылающиеся на объекты, имеют специальный тип: **менеджер объектов**. До сих пор мы работали с менеджером объектов `objects`, делая запросы вида `User.objects.get(id=2)` или `Post.objects.all()`.

Свойство `posts` — такой же менеджер объектов, как и `objects`, разница лишь в том, что оно было создано при связывании моделей.

Для менеджеров объектов можно вызывать метод `count()`, который пересчитает связанные объекты. Django сам разберется, как правильно составить запрос в этом случае.

Чтобы применить агрегирующие функции к связанным данным из других таблиц, запрос делается через **аннотирование**, методом `annotate()`.

Метод `annotate()`

Чтобы получить количество записей, созданных каждым пользователем, нужно сгруппировать несколько таблиц и добавить объекту новое свойство,

которое будет содержать количество связанных с ним объектов в другой таблице.

Конечно, можно было бы в цикле пройтись по всем пользователям, вызывая метод `count()` менеджера объектов `posts`. Но это создало бы огромную нагрузку на базу и каскад однотипных запросов.

В чистом SQL вопрос решается так: в результирующую выборку попадают запрошенные столбцы исходных таблиц и дополнительные столбцы с результатами вычислений.

Похожим образом работает метод `annotate()` в ORM, но в результате к полученным объектам **добавляется новое свойство**, содержащее результат вычисления.

В следующем примере аргумент `posts_count` — это имя нового свойства объекта, оно появится у объектов модели `User`:

```
# Достать из модели User все объекты,
# создать свойство posts_count и записать в него число постов, связанных с
автором.
# posts — это свойство модели User, менеджер объектов
>>> annotated_results = User.objects.annotate(posts_count =
Count('posts'))
>>
>> annotated_results
(0.001) SELECT "auth_user"."id", "auth_user"."password",
"auth_user"."last_login", "auth_user"."is_superuser", "auth_user"."username",
"auth_user"."first_name", "auth_user"."last_name", "auth_user"."email",
"auth_user"."is_staff", "auth_user"."is_active", "auth_user"."date_joined",
COUNT("posts_post"."id") AS "posts_count" FROM "auth_user" LEFT OUTER JOIN
"posts_post" ON ("auth_user"."id" = "posts_post"."author_id") GROUP BY
"auth_user"."id", "auth_user"."password", "auth_user"."last_login",
"auth_user"."is_superuser", "auth_user"."username", "auth_user"."first_name",
"auth_user"."last_name", "auth_user"."email", "auth_user"."is_staff",
"auth_user"."is_active", "auth_user"."date_joined" LIMIT 21; args=()
<QuerySet [<User: admin>, <User: Leo>]>

# перебрать в цикле список пользователей annotated_results
# и для каждого объекта вывести свойство name
# и новое свойство posts_count, которое хранит число постов пользователя
>>> for item in annotated_results:
...     print(f"Постов у пользователя {item.username}:
{item.posts_count}")
...
(0.000) SELECT "auth_user"."id", "auth_user"."password",
"auth_user"."last_login", "auth_user"."is_superuser", "auth_user"."username",
"auth_user"."first_name", "auth_user"."last_name", "auth_user"."email",
"auth_user"."is_staff", "auth_user"."is_active", "auth_user"."date_joined",
COUNT("posts_post"."id") AS "posts_count" FROM "auth_user" LEFT OUTER JOIN
```

```
"posts_post" ON ("auth_user"."id" = "posts_post"."author_id") GROUP BY  
"auth_user"."id", "auth_user"."password", "auth_user"."last_login",  
"auth_user"."is_superuser", "auth_user"."username", "auth_user"."first_name",  
"auth_user"."last_name", "auth_user"."email", "auth_user"."is_staff",  
"auth_user"."is_active", "auth_user"."date_joined"; args=()  
Постов у пользователя admin: 2  
Постов у пользователя leo: 36  
# хорошо, что у нас пока что не 100500 авторов
```

Разница между annotate() и aggregate()

Метод `annotate()` возвращает объекты и добавляет к ним новые свойства:

```
>>> rez = User.objects.annotate(written_posts = Count('posts'))  
>>> rez[1].written_posts  
36  
# у объекта класса User появилось свойство written_posts,  
# хотя в модели User оно не описано
```

Метод `aggregate()` отдает только значение, результат работы агрегирующей функции:

```
>>> Checks.objects.aggregate(average_price=Avg('price'))  
{'average_price': 127.01}
```

19_Управление пользователями

Нашему сайту необходима система для регистрации и авторизации пользователей. В Django есть встроенные инструменты для работы с пользователями, и мы задействуем эту часть фреймворка в нашем проекте.

В интерфейсе администратора уже есть подраздел управления пользователями. Через командную строку вы создали суперпользователя, а при обновлении БД в системе появился пользователь *leo*. Но было бы странно добавлять остальных пользователей вручную. Дадим им возможность регистрироваться самостоятельно.

Модуль django.contrib.auth

Исходный код модулей, пакетов и библиотек вашего проекта сохранён в папке */venv*. В ней находится виртуальное окружение проекта. Если вы хотите узнать, как работает проект — почитайте исходный код и документацию.

В состав исходного кода фреймворка Django входит директория `/contrib` — «склад полезных вещей», набор готовых приложений для решения стандартных задач: поддержки мультистивости, страниц flatpages, работы с ГИС, системы site map, публикации RSS-лент etc. Любой из этих приложений можно подключить по определённым правилам к проекту — и оно заработает «из коробки».

Приложение `django.contrib.auth` было установлено автоматически вместе с Django при подготовке окружения. Именно оно отвечает за работу с пользователями. Посмотрим, как оно устроено и как с ним работать. Для начала найдём основные элементы этого приложения.

Поиск URL-шаблонов

В модуле `django.contrib.auth` есть файл `urls.py`. Посмотрите, какие адреса станут доступны на нашем сайте после подключения этого модуля:

```
from django.contrib.auth import views
from django.urls import path

urlpatterns = [
    path('login/', views.LoginView.as_view(), name='login'),
    path('logout/', views.LogoutView.as_view(), name='logout'),

    path('password_change/', views.PasswordChangeView.as_view(),
name='password_change'),
    path('password_change/done/', views.PasswordChangeDoneView.as_view(),
name='password_change_done'),

    path('password_reset/', views.PasswordResetView.as_view(),
name='password_reset'),
    path('password_reset/done/', views.PasswordResetDoneView.as_view(),
name='password_reset_done'),
    path('reset/<uidb64>/<token>/',
views.PasswordResetConfirmView.as_view(), name='password_reset_confirm'),
    path('reset/done/', views.PasswordResetCompleteView.as_view(),
name='password_reset_complete'),
]
```

Обратите внимание на имена (параметр `name`) для URL-шаблонов списка `urlpatterns`. По этим именам можно будет обращаться к страницам, в дальнейшем нам это пригодится.

View-классы приложения Auth

Если посмотреть исходный код модуля `django/contrib/auth/views.py` (локально этот файл доступен в директории виртуального окружения проекта `venv/lib/`

`python3.8/site-packages/django/contrib/auth/views.py`), то мы найдём множество *Class Based View*.

Class Based View — это классы, по своему назначению аналогичные view-функциям, они так же обрабатывают запросы и возвращают ответ.

Поищите в файле названия классов, в имени которых есть слово "View", и найдите в них свойство `template_name`: это предустановленные названия шаблонов, необходимых для работы с пользователями

```
class LoginView(SuccessURLAllowedHostsMixin, FormView):
    template_name = 'registration/login.html'
class LogoutView(SuccessURLAllowedHostsMixin, TemplateView):
    template_name = 'registration/logged_out.html'
class PasswordResetView(PasswordContextMixin, FormView):
    template_name = 'registration/password_reset_form.html'
class PasswordResetDoneView(PasswordContextMixin, TemplateView):
    template_name = 'registration/password_reset_done.html'
class PasswordResetConfirmView(PasswordContextMixin, FormView):
    template_name = 'registration/password_reset_confirm.html'
class PasswordResetCompleteView(PasswordContextMixin, TemplateView):
    template_name = 'registration/password_reset_complete.html'
class PasswordChangeView(PasswordContextMixin, FormView):
    template_name = 'registration/password_change_form.html'
class PasswordChangeDoneView(PasswordContextMixin, TemplateView):
    template_name = 'registration/password_change_done.html'
```

Давайте разберёмся, какими страницами управляют эти view-классы:

- **LoginView** — страница с формой авторизации;
- **LogoutView** — страница выхода, дающая пользователю возможность прекратить работу с сайтом;
- **PasswordResetView** — страница восстановления пароля, здесь можно ввести свой email и получить ссылку для восстановления доступа;
- **PasswordResetDoneView** — страница уведомления о том, что ссылка на восстановление пароля отправлена;
- **PasswordChangeView** — эта страница будет доступна по ссылке при восстановлении пароля, здесь пользователь сможет задать новый пароль;
- **PasswordChangeDoneView** — страница уведомления о том, что пароль изменён.

В этом списке не хватает страницы регистрации, её мы создадим отдельно.

В директории для шаблонов модуля Auth, `venv/lib/python3.8/site-packages/django/contrib/auth/templates/` этих файлов нет, их нужно создать самостоятельно.

Надо будет создать такие шаблоны для нашего проекта Poemnotes:

- `registration/login.html`
- `registration/logged_out.html`
- `registration/password_reset_form.html`
- `registration/password_reset_done.html`
- `registration/password_reset_confirm.html`
- `registration/password_change_form.html`
- `registration/password_change_done.html`
- И еще `signup.html` для регистрации новых пользователей

Создание отдельного приложения Users

Для того чтобы собрать весь код для управления регистрацией пользователей в одном месте, самостоятельно создайте новое приложение **Users** и добавьте его в начало списка `INSTALLED_APPS` в конфиге сайта.

Скорее всего, вам пригодится консольная команда

```
$ python manage.py startapp
```

20_Generic Views

Разновидности view

При обращении к какому-нибудь URL специальный обработчик `path()` вызывает **объект**, передавая ему в качестве аргумента объект типа `request`, а на выходе ожидает объект типа `response`. Вызываемым объектом может быть view-функция, но можно вызвать классы и их методы.

- **View-функция** — такие функции называют ещё «функции представления» (от слова "view") или просто «представление». Это самый простой тип view-объектов. Функция получает на вход стандартный объект `request` и возвращает объект типа `response`. Объекты `response` могут быть созданы встроенными функциями-помощниками, например, функцией `render()`. Вызов view-функции: `path('view/', my_view)`
- **Class-based view** — по аналогии название можно перевести как «представление, основанное на классе» или «представление-класс». Мы будем использовать термин «view-класс». Такой класс должен наследоваться от специальных родительских классов **Generic Views**. Из файлов `urls.py` можно вызывать метод view-класса `as_view()`. Вызов метода view-класса: `path('view/', ClassName.as_view())`

Объект, который можно вызывать, называют «исполняемый», *callable*.

Generic Views

Generic Views — это встроенные в Django view-классы, созданные для решения стандартных задач. В переводе *Generic Views* — это «общий вид» или «базовое представление». Популярные *Generic Views*:

- **FormView** обрабатывает формы на основе моделей.
- **TemplateView** упрощает вывод данных в шаблон.
- **CreateView** связывает модель и пользовательскую веб-форму, предназначенную для создания новой записи в базе.

На основе *Generic Views* создают классы-наследники: view-классы, обладающие свойствами и методами *Generic Views*; это классическое ООП в действии.

Связь модели, формы и view-класса

Модель — это класс для работы с ORM, посредник между кодом и базой данных. Для свойств модели указывают типы данных, и на основе модели можно создать форму, поля которой будут соответствовать свойствам модели.

Такая форма даст пользователю возможность добавлять или изменять записи в базе данных.

Создание форм на основе моделей:

- Создаётся (или выбирается существующая) модель.
- На основе модели создаётся **класс формы**.
- Объект формы (экземпляр, созданный на основе класса формы) передаётся в специальный view-класс.
- View-класс передаёт объект формы в шаблон, создаёт и возвращает пользователю страницу с веб-формой.

Для создания форм на основе моделей есть предустановленный класс **ModelForm**: от него можно наследовать классы для генерации форм.

Вот пример создания формы через **ModelForm**:

```
from django.db import models
from django.forms import ModelForm

# создадим модель, в которой будем хранить данные формы
class Book(models.Model):
    name = models.CharField(max_length=100)
    isbn = models.CharField(max_length=100)
    pages = models.IntegerField(min_value=1)

class BookForm(ModelForm):
    class Meta:
        # эта форма будет работать с моделью Book
        model = Book
        # на странице формы будут отображаться поля 'name', 'isbn' и
`pages`
        fields = ['name', 'isbn', 'pages']
```

Создана модель **Book**. Следом за ней описан класс формы **BookForm**, он наследуется от **ModelForm**.

В **BookForm** создан конфигурационный блок `class Meta`, в нём указана модель, на основе которой должна быть создана форма **BookForm**: `model = Book`.

У модели **Book** есть три свойства, два из которых принимают строку, а третье — число. Поэтому HTML-форма будет иметь поля того же типа, и мы ожидаем, что она будет примерно такой

A screenshot of a Django form interface. It contains three fields: 'name' (a standard text input field), 'isbn' (a standard text input field), and 'pages' (a number input field with up and down arrow buttons to increment or decrement the value).

Теперь надо вывести эту форму на страницу: передать её во view-функцию или view-класс. Это стандартная задача, и её тоже упростили: в Django для этого есть отдельный **Generic View CreateView**.

```
from django.views.generic import CreateView

class BookView(CreateView):
    form_class = BookForm
    success_url = "/thankyou" # куда переадресовать пользователя после
    успешной отправки формы
    template_name = "new_book.html"
```

В HTML-шаблон передаётся переменная `form`:

new_book.html

```
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Отправить">
</form>
```

urls.py

```
# теперь из файла urls.py для пути new_book/
# можно вызвать метод as_view() класса BookView
# этот метод унаследован классом BookView от родителя
urlpatterns = [
    # ...
    path("new_book/", views.BookView.as_view(), name="new_book")
]
```

Данные, введённые пользователем в форму, после отправки пройдут примерно такой путь:

HTML-форма → класс формы → модель → ORM → БД

Готово: в базу данных можно добавлять новые записи со страницы сайта.

База книг и форма для её заполнения в проекте **Poemnotes** не нужны, так что в следующем уроке вы создадите форму для регистрации пользователей: принципы всё те же, а пользы больше.

21_Добавление формы регистрации

Сейчас будет немного теории, цепочка классов — и вы научитесь создавать формы для работы с данными в базе.

Вот план дальнейшей работы:

- На основе встроенного класса **UserCreationForm** напишем класс **CreationForm**, он создаст объект формы регистрации, данные из которой будут передаваться в модель **User**.
- На основе *Generic Views* **CreateView** создадим view-класс **SignUp**, который вызовет шаблон и передаст в него форму **CreationForm**.
- Создадим HTML-шаблон, он примет словарь `form` из view-класса **CreateView**.
- Добавим вызов view-класса **SignUp** в *urls.py*.

Создание формы на основе класса **UserCreationForm**

С технической точки зрения «процесс регистрации» — это создание нового объекта модели **User**. Пользователь отправляет через форму свои данные, после проверки эти данные передаются в модель **User** и сохраняются в базе.

Регистрация пользователя — стандартная и востребованная процедура, и авторы Django сделали за нас основную работу. В модуле *django/contrib/auth/forms.py* для создания формы регистрации заготовлен класс **UserCreationForm** (наследник знакомого вам встроенного класса **ModelForm**), на основе которого **создаётся форма регистрации**.

django/contrib/auth/forms.py

```
class UserCreationForm(forms.ModelForm):  
    """
```

```
A form that creates a user, with no privileges, from the given username  
and  
password.  
....  
# ...
```

В исходном коде класса **UserCreationForm** видно, что форма создается на основе модели **User**. Класс **UserCreationForm** (как и любые наследники класса **ModelForm**) считывает и добавляет свойства модели как поля формы.

Сейчас мы создадим класс **CreationForm**, наследника класса **UserCreationForm**. Классу **ModelForm** он будет приходиться внуком: **ModelForm → UserCreationForm → CreationForm**.

Класс **CreationForm** можно было бы и не создавать, а напрямую подключить встроенный класс **UserCreationForm** из пакета **django.contrib.auth**, но для нашей работы нужно внести изменения в работу предустановленного класса: хочется вывести на страницу не все поля, а лишь те, которые нужны для регистрации именно на нашем сайте.

В Django принято хранить формы в отдельном файле, и мы последуем этому правилу.

Создайте шаблон страницы регистрации `users/forms.py` и добавьте в него код.

```
from django.contrib.auth.forms import UserCreationForm  
from django.contrib.auth import get_user_model  
  
User = get_user_model()  
  
# создадим собственный класс для формы регистрации  
# сделаем его наследником предустановленного класса UserCreationForm  
class CreationForm(UserCreationForm):  
    class Meta(UserCreationForm.Meta):  
        # укажем модель, с которой связана создаваемая форма  
        model = User  
        # укажем, какие поля должны быть видны в форме и в каком порядке  
        fields = ("first_name", "last_name", "username", "email")
```

Согласно рекомендациям разработчиков Django, к модели **User** лучше обращаться через функцию `get_user_model()`. Это нужно для того, чтобы разработчик без труда мог переопределить модель, которая будет хранить

данные пользователей. По умолчанию это модель **User**, она создаётся при установке Django. Но если она вас не устраивает — вы можете дополнить базовую модель, унаследовавшись от **User**, описать свойства новой модели и зарегистрировать её в системе в качестве модели пользователей.

Функция **get_user_model()** обращается именно к той модели, которая зарегистрирована в качестве основной модели пользователей в конфиге проекта.

Если разработчик заменит эту модель на собственную, вносить изменения по всему проекту ему не придётся, будет достаточно изменить лишь одно значение в конфиге. Но это в случае, если он повсюду предусмотрительно применял **get_user_model()**.

Наследственность в классе **Meta**

В классе **CreationForm** описан вложенный класс **Meta**: он унаследован от родительских классов. В нём настраивается форма, и именно в нём мы переопределяем некоторые её параметры.

Конструкция `class Meta(UserCreationForm.Meta)` описывает обычное наследование, только наследуется не основной класс, а вложенный:

```
# наследуется класс UserCreationForm:  
class CreationForm(UserCreationForm):  
# наследуется класс Meta, вложенный в класс UserCreationForm:  
    class Meta(UserCreationForm.Meta):  
# ...
```

В результате класс **Meta**, вложенный в класс **CreationForm**, унаследует все ключи `UserCreationForm.Meta`, но теперь мы получаем возможность переопределить их. Копировать целиком код класса **Meta** из `UserCreationForm` в свой код было бы не лучшей идеей, такой подход грозит проблемами: если в следующей версии Django в `UserCreationForm.Meta` добавят что-то новое — придется искать и исправлять места, где возникла несовместимость. Наследование всегда лучше.

Обратите внимание: в исходном классе `UserCreationForm` в классе `Meta` есть строка

```
model = User
```

Ссылка идёт прямо на модель **User**, без посредства функции `get_user_model()`. В своём коде, в классе **CreationForm**, мы переопределяем переменную `model`, присвоив ей значение `get_user_model()` (передав его через переменную `User`).

Отображение формы

Создайте в файле `users/views.py` view-класс **SignUp**, унаследовав его от *Generic View CreateView*:

```
# импортируем CreateView, чтобы создать ему наследника
from django.views.generic import CreateView

# функция reverse_lazy позволяет получить URL по параметру "name" функции
path()
# берём, тоже пригодится
from django.urls import reverse_lazy

# импортируем класс формы, чтобы сослаться на неё во view-классе
from .forms import CreationForm

class SignUp(CreateView):
    form_class = CreationForm
    success_url = reverse_lazy("login") # где login – это параметр "name" в
path()
    template_name = "signup.html"
```

Нам остаётся только определить некоторые параметры конфигурации, и view-класс **CreateView** сам отрисует форму.

- **form_class** — из какого класса взять форму
- **success_url** — куда перенаправить пользователя после успешной отправки формы
- **template_name** — имя шаблона, куда будет передана переменная `form` с объектом HTML-формы. Всё это чем-то похоже на вызов функции `render()` во view-функции.

Теперь в шаблон `signup.html` будет выведена форма, описанная в классе **CreationForm**. После отправки этой формы пользователь будет переадресован на страницу, для которой в `urls.py` указано имя `name="login"`. Данные, отправленные через форму, будут переданы в модель **User** и сохранены в БД.

Готово всё, кроме шаблона `signup.html`.

Добавление шаблона

Для удобства организации кода создайте директорию `users/templates`. Это будет директория шаблонов приложения `Users`.

Django будет работать с такими директориями, если в `settings.py` в директиве `TEMPLATES` для ключа `APP_DIRS` установить `True`.

После установки этого ключа Django будет искать шаблоны не только в головной директории `templates`, но и в папках `templates` в директориях приложений (если такие папки там есть).

В `users/templates` создайте файл `signup.html` и добавьте в него код для отображения формы:

```
{% extends "base.html" %}  
{% block title %}Зарегистрироваться{% endblock %}  
{% block content %}  
  
<form method="post" action="{% url 'signup' %}">  
    {% csrf_token %}  
    {{ form.as_p }}  
    <input type="submit" value="Зарегистрироваться">  
</form>  
{% endblock %}
```

Добавление страницы регистрации в `urls.py`

Для адресов страниц, относящихся к регистрации и входу на сайт, мы будем использовать префикс `auth/`.

Подключите файлы `urls.py` приложений **Users** и **Auth** к головному `urls.py` по аналогии с уже подключённым `posts.urls`.

После изменений ваш файл `poemnotes/urls.py` должен выглядеть так:

```
from django.contrib import admin  
from django.urls import include, path  
  
urlpatterns = [  
    # обработчик для главной страницы ищем в urls.py приложения posts  
    path("", include("posts.urls")),  
  
    # регистрация и авторизация  
    path("auth/", include("users.urls")),
```

```

# если нужного шаблона для /auth не нашлось в файле users.urls -
# ищем совпадения в файле django.contrib.auth.urls
path("auth/", include("django.contrib.auth.urls")),

# раздел администратора
path("admin/", admin.site.urls),
]

```

Теперь добавьте в файл *users/urls.py* адрес страницы регистрации пользователей:

```

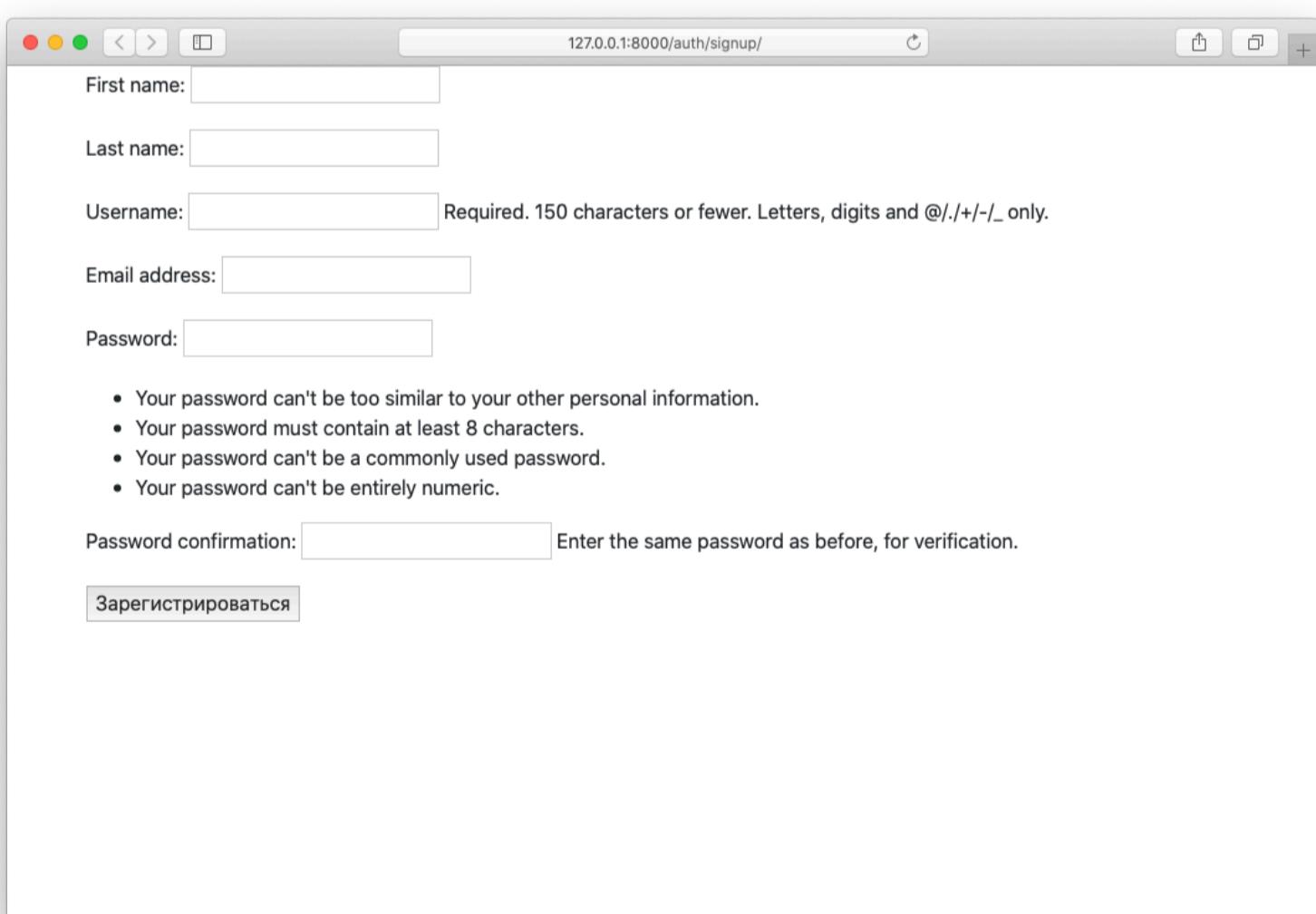
from django.urls import path
from . import views

urlpatterns = [
    # path() для страницы регистрации нового пользователя
    # её полный адрес будет auth/signup/, но префикс auth/ обрабатывается в
    # головном urls.py
    path("signup/", views.SignUp.as_view(), name="signup")
]

```

Всё!

Зайдите на страницу <http://127.0.0.1:8000/auth/signup/>. Там должна отобразиться приблизительно такая форма:



Готово, всё работает, хоть и выглядит не очень нарядно.

Следующий урок посвятим наведению красоты.

22_Доработка базового шаблона

Изменим основной шаблон: добавим в него шапку сайта с названием проекта и навигацией. В качестве основы мы будем использовать примеры, которые предлагает [сайт с документацией по Bootstrap](#).

В интернете есть много бесплатных тем оформления, расширяющих стандартные возможности Bootstrap. Будет хорошо, если вы найдёте время и оформите сайт по своему вкусу, а заодно немного разберётесь с HTML.

Формирование контекста шаблона

Если пользователь ещё не залогинился, в шапке сайта предложим ему ссылки на страницы входа и регистрации. А если он уже вошёл — дадим ему ссылки на страницы смены пароля и выхода. Для этого надо передать в шаблон переменную с информацией о том, авторизирован ли пользователь.

Обратите внимание на переменную `TEMPLATES` в конфигурационном файле `settings.py`:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [TEMPLATES_DIR],
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
# вот это интересно
                "django.contrib.messages.context_processors.messages",
            ]
        },
    }
]
```

Перед обработкой шаблона вызываются функции из списка `context_processors`. По мере исполнения эти функции могут

добавлять переменные, доступные при обработке шаблона. Другими словами, эти функции изменяют контекст шаблона.

Функция `django.contrib.auth.context_processors.auth` добавит в контекст шаблона переменную `user`. Она может быть либо объектом типа `AnonymousUser`, либо экземпляром модели `User`.

Если заглянуть в файл `django/contrib/auth/context_processors.py` и найти там функцию `auth`, то видно, что она возвращает словарь с ключами `user` и `perms`.

Добавление шапки и подвала сайта

Теперь понятно, как изменять шапку сайта в зависимости от того, залогинен ли пользователь: надо проверить состояние переменной `user`. Создайте файл `templates/nav.html` с таким содержимым:

```
<nav class="navbar navbar-light" style="background-color: #e3f2fd;">
    <a class="navbar-brand" href="/"><span style="color: red">Ya</span>tube</a>
    <nav class="my-2 my-md-0 mr-md-3">
        {% if user.is_authenticated %}
            Пользователь: {{ user.username }}.
            <a class="p-2 text-dark" href="{% url 'password_change' %}">Изменить
пароль</a>
            <a class="p-2 text-dark" href="{% url 'logout' %}">Выйти</a>
        {% else %}
            <a class="p-2 text-dark" href="{% url 'login' %}">Войти</a> |
            <a class="p-2 text-dark" href="{% url 'signup' %}">Регистрация</a>
        {% endif %}
    </nav>
</nav>
```

Подключите файл `templates/nav.html` к базовому шаблону `templates/base.html` тегом `{% include %}`:

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">
        <title>{% block title %}The Last Social Media You'll Ever Need{% endblock %} | Poemnotes</title>
        <!-- Загрузка статики -->
        {% load static %}
        <link rel="stylesheet" href="{% static 'bootstrap/dist/css/
bootstrap.min.css' %}">
        <script src="{% static 'jquery/dist/jquery.min.js' %}"></script>
        <script src="{% static 'bootstrap/dist/js/bootstrap.min.js' %}"></script>
```

```

</head>
<body>
    {% include 'nav.html' %}
    <main>
        <div class="container">
            {% block content %}
            <!-- Содержимое страницы -->
            {% endblock content %}
        </div>
    </main>

    </body>
</html>

```

Тем же способом добавьте файл шаблона для блока, отображаемого внизу страницы, его называют «подвал» или "footer". Назовём его footer.html:

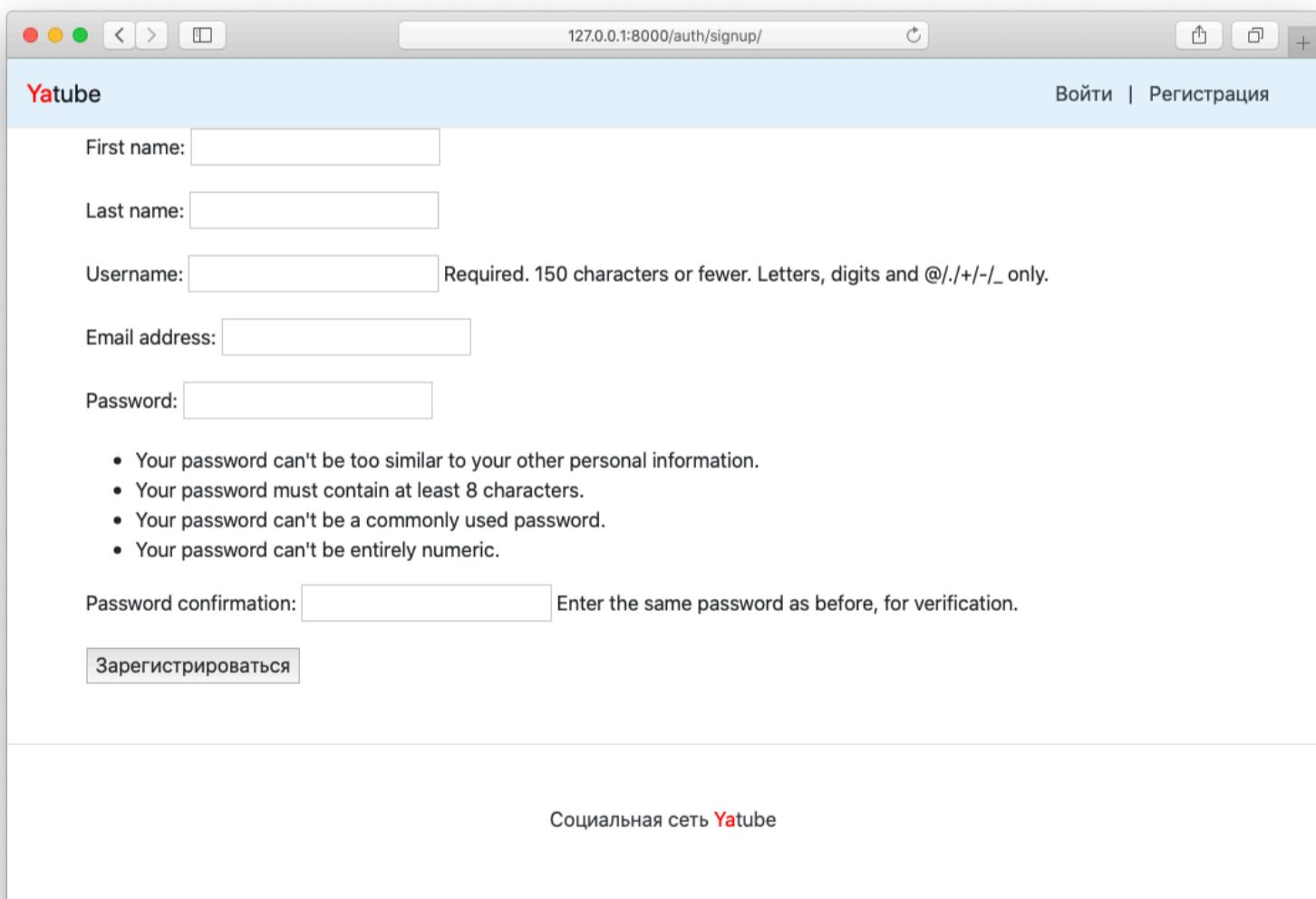
```

<footer class="pt-4 my-md-5 pt-md-5 border-top">
    <p class="m-0 text-dark text-center">Социальная сеть <span
style="color:red">Ya</span>tube </p>
</footer>

```

Добавьте вызов этого блока после закрывающего тега `<main>` базового шаблона.

Обновите страницу в браузере. Страница должна измениться. Если вы не авторизованы, она будет выглядеть так:



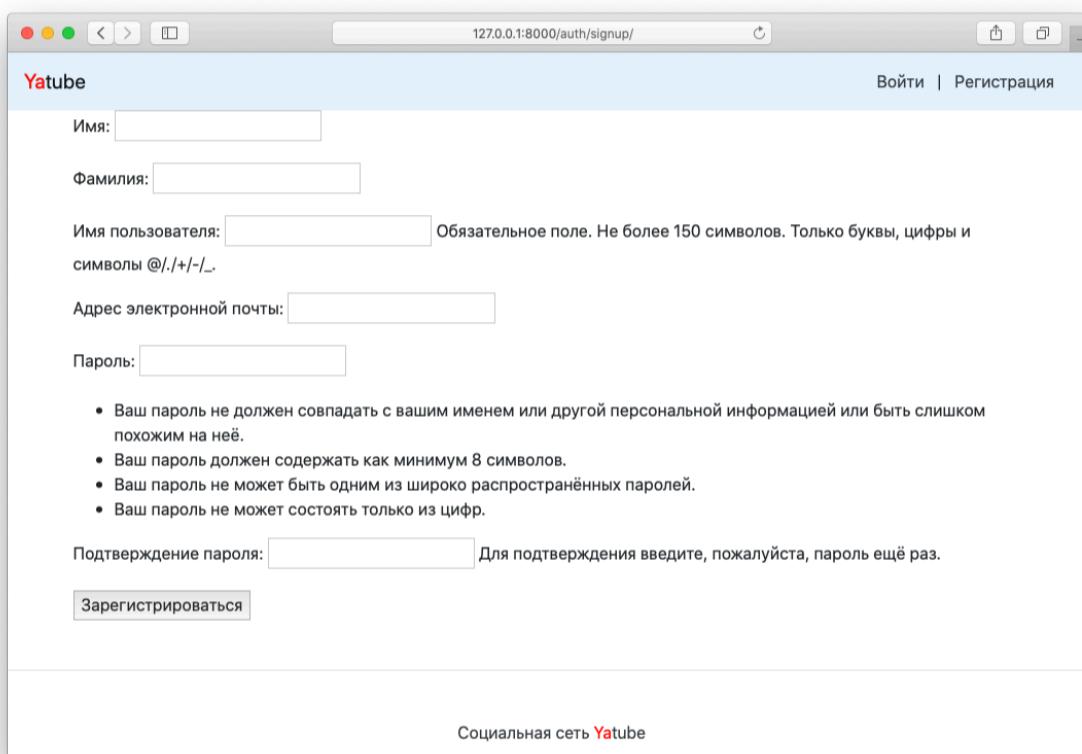
Ссылка «Войти» пока что будет выдавать ошибку: шаблон для страницы входа ещё не создан. Ссылка «Регистрация» должна сработать нормально: она отправит вас на страницу /signup.

23_Язык сайта

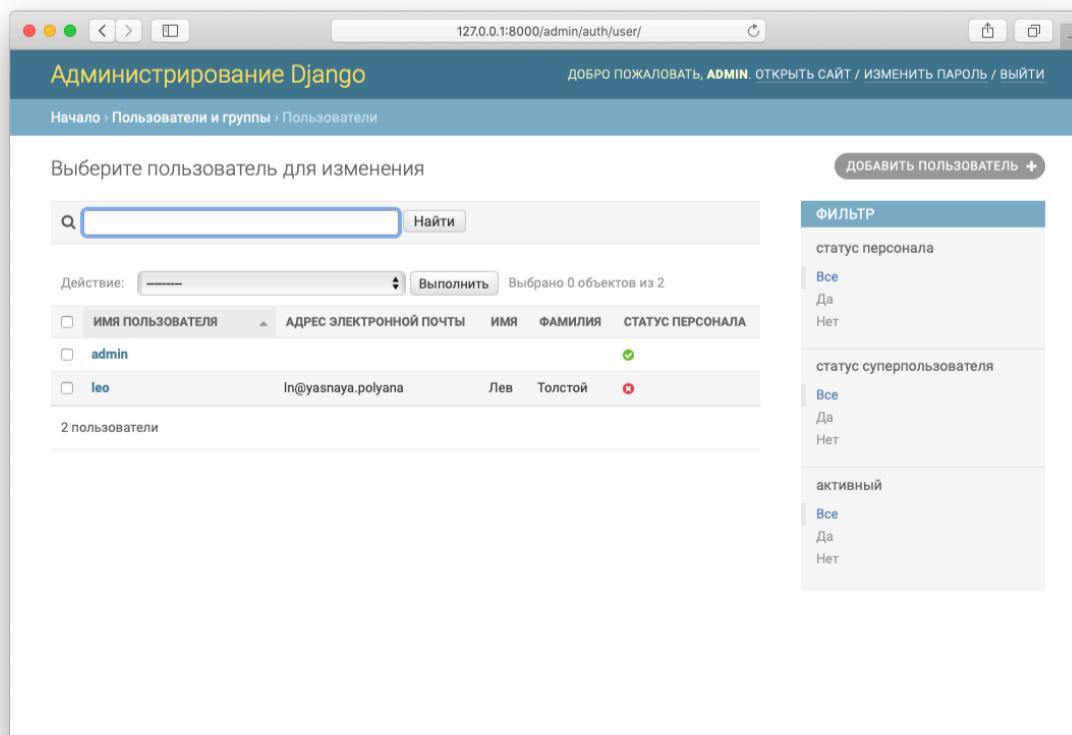
Измените значение переменной LANGUAGE_CODE в файле `settings.py`:

```
LANGUAGE_CODE = "ru"
```

Зайдите на страницу регистрации:



Также изменится интерфейс администратора, система заговорит по-русски:



В Django встроена система интернационализации. В рамках этого курса мы не будем с ней работать, но применим встроенные инструменты фреймворка для того, чтобы перевести интерфейс на русский язык.

Django поддерживает достаточно большое количество языков. Если вы хотите перевести интерфейс на другой язык, вам надо найти код языка в таблице на сайте, убедиться, что он поддерживается Django, и подставить код языка в переменную **LANGUAGE_CODE**.

24_Доработка шаблона формы

В этом уроке мы продолжим работу с формой регистрации. Шаблон `users/templates/signup.html` сейчас выглядит так:

```
{% extends "base.html" %}  
{% block title %}Зарегистрироваться{% endblock %}  
{% block content %}  
  
<form method="post" action="{% url 'signup' %}">  
    {% csrf_token %}  
    {{ form.as_p }}  
    <input type="submit" value="Зарегистрироваться">  
</form>  
{% endblock %}
```

Переменная `form` передаётся в шаблон, на её основе генерируется HTML-код формы. Метод `as_p()` выводит код формы, обрамляя каждую строку формы в HTML-тег `<p>`. Аналогичные методы `as_ul()` и `as_table()` выведут форму, обернув её в HTML-код маркированного списка `` или таблицы `<table>` соответственно. Чуть ниже мы покажем примеры.

Что такое `{% csrf_token %}` ?

CSRF (от англ. *cross-site request forgery* — межсайтовая подделка запроса) — вид атаки на сайт или на аккаунт пользователя на сайте, где пользователь авторизован. Логика подобной атаки довольно проста:

- предположим, пользователь залогинен на сайте своего банка
- пользователь заходит на некую стороннюю веб-страницу и нажимает на заражённую ссылку
- при клике отправляется запрос на сайт банка (и так совпало, что это именно банк нашего пользователя)

- браузер помнит данные пользователя, и поэтому отправленный запрос будет воспринят сайтом банка как запрос авторизованного пользователя
- посредством такого запроса злоумышленники могут вывести деньги с аккаунта пользователя или сделать заказ от его имени

Конечно, в этом сценарии должен присутствовать элемент невезения: запрос должен отправляться на сайт именно того банка, где залогинен пользователь. Но если банк популярен, а число пользователей, кликающих на хакерскую ссылку, велико — такая схема сработает.

Для защиты от таких атак сайты генерируют специальный **csrf-токен** (или *csrf-ключ*), который встраивается в «доверенную» веб-страницу и отправляется вместе с каждым запросом от пользователя к серверу. Этот ключ — уникальная для каждого пользователя строка, последовательность символов. Угадать её практически невозможно.

При получении запроса сервер сравнивает ключ, полученный с запросом, с образцом, сохранённым на сервере, и обрабатывает запрос только в случае совпадения.

Django серьёзно относится к безопасности, и все формы с POST-запросами по умолчанию должны быть защищены таким ключом.

Работа с формами

Обычно HTML-форма выглядит приблизительно так:

```
<form action="/example-action" method="POST">
    Введите имя:<br>
    <input type="text" name="firstname">
    <br>
    Введите фамилию:<br>
    <input type="text" name="lastname">
    <br><br>
    <input type="submit" value="Отправить">
</form>
```

В этой форме есть два поля (Имя, Фамилия) и кнопка «Отправить». При нажатии на кнопку браузер отправит данные из формы на страницу, указанную в атрибуте **action**, в нашем примере это /example-action.

Атрибут **method** определяет способ отправки данных. Если Лев Николаевич Толстой решит заполнить форму и будет честен, то при отправке данных

методом GET браузер обратится к странице `/action?firstname=Лев&lastname=Толстой`, а метод POST отправит значение полей в теле HTTP запроса.

Валидация форм

При отправке форм могут возникнуть разнообразные ошибки. Пользователь может случайно нажать кнопку отправки формы до её заполнения, может ввести почтовый адрес с ошибкой и обязательно найдётся человек, который сделает опечатку при вводе пароля — не зря при регистрации сайты просят ввести пароль дважды. Проверка форм на корректность заполнения называется **валидация**.

Устройство форм в Django

Чтобы было проще понимать примеры и код, разберём устройство форм на высоком уровне.

Форма в Django описывается в классе, похожем на модель. Класс должен быть унаследован от встроенного класса `Form`:

```
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(label="Введите имя")
    sender = forms.EmailField(label="Email для ответа")
    subject = forms.CharField(label="Тема сообщения", initial='Письмо
администратору', max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    cc_myself = forms.BooleanField(label="Отправить себе копию",
required=False)
```

Форма состоит из полей разных типов, все они [описаны в документации](#). Когда в шаблоне поле превращается в HTML-код, то используется виджет, определённый параметром `widget`. **Виджет** — это шаблон, по которому генерируется HTML-код поля формы.

Основные типы полей, которые вам будут встречаться:

- **BooleanField** — соответствует типу `bool`. Виджет по умолчанию отрисовывает чекбокс `<input type="checkbox">`
- **CharField** — поле для ввода текста, по умолчанию используется виджет односторочного поля ввода `<input type="text">`. Виджет можно заменить: если указать в параметрах `widget=forms.Textarea`, будет отрисовано поле многострочного ввода, `<textarea>`

- **ChoiceField** – поле выбора из выпадающего списка, `<select>`
- **EmailField** – одностороннее поле ввода текста, но с обязательной проверкой введённой строки на соответствие формату email
- **FileField** – поле для отправки файла, в шаблоне отрисует тег `<input type="file">`. Есть аналогичное поле для отправки только файлов изображений: **ImageField**
- **IntegerField** – поле для ввода чисел: `<input type="number">`

| | |
|---|--|
| <p>Текстовое поле</p> <input type="text" value="Ваше имя"/> | <p>Многострочное текстовое поле</p> <pre>Не печалься, друг мой нежный, будет нам еще с тобой.</pre> |
| <p>Кнопка</p> <input type="button" value="Submit"/> | <p>Флажки</p> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> |
| <p>Список</p> <input style="width: 100px; height: 20px; border: 1px solid #ccc; padding: 2px;" type="button" value="Чай"/> | <p>Переключатели</p> <input checked="" type="radio"/> <input type="radio"/> <input type="radio"/> |

Можно самостоятельно создавать новые типы полей и новые виджеты. В Django есть множество готовых виджетов, например, для превращения поля ввода в визуальный редактор.

Работа с формами из кода

Запустите в консоли интерактивный режим Django, дальше работать будем в нём: (venv) \$ `python manage.py shell`

Импортируем модуль `forms` и создадим класс формы с двумя полями:

```
>>> from django import forms
>>> class Registration(forms.Form):
...     firstname = forms.CharField(label="Введите имя", initial='Лев')
...     lastname = forms.CharField(label="Введите фамилию", initial='Толстой')
...
>>> form = Registration()
# напечатаем результат, чтобы увидеть HTML-код, который выведет метод as_p()
>>> print(form.as_p())
<p><label for="id_firstname">Введите имя:</label> <input type="text"
name="firstname" value="Лев" required id="id_firstname"></p>
<p><label for="id_lastname">Введите фамилию:</label> <input type="text"
name="lastname" value="Толстой" required id="id_lastname"></p>
```

Сгенерированный HTML-код содержит код полей ввода `<input type="text" ...>` с необходимыми атрибутами и теги `<label>` — заголовки полей, видимые пользователям.

Метод `as_p()`, унаследованный от класса `Form`, обрамляет каждую пару тегов «`label + поле`» в HTML-тег `<p>`

Ведите имя:

Ведите фамилию:

По умолчанию форма выводится в HTML-таблицу, в элемент `<table>`. Такой же код будет выведен и методом `as_table()`:

```
>>> print(form.as_table())
<tr><th><label for="id_firstname">Ведите имя:</label></th><td><input
type="text" name="firstname" value="Лев" required id="id_firstname"></td></tr>
<tr><th><label for="id_lastname">Ведите фамилию:</label></th><td><input
type="text" name="lastname" value="Толстой" required id="id_lastname"></td></
tr>
```

Ведите имя:

Ведите фамилию:

Вывод списком, методом `as_ul()`:

```
>>> print(form.as_ul())
<li><label for="id_firstname">Ведите имя:</label> <input type="text"
name="firstname" value="Лев" required id="id_firstname"></li>
<li><label for="id_lastname">Ведите фамилию:</label> <input type="text"
name="lastname" value="Толстой" required id="id_lastname"></li>
```

- Ведите имя:
- Ведите фамилию:

Каждый из этих методов можно вызывать из шаблона командами `form.as_table` , `form.as_p` или `form.as_ul` .

Когда форма заполнена и отправлена, Django получит данные и проверит их на корректность. В случае, если отправленная информация не прошла валидацию, то объект `form` получит список ошибок в атрибуте `{{ form.errors }}`.

Работа с полями формы

С объектом формы можно работать через цикл `for`:

```
>>> for field in form:  
...     print(field)  
...  
# поля формы будут напечатаны по очереди  
<input type="text" name="firstname" value="Лев" required id="id_firstname">  
<input type="text" name="lastname" value="Толстой" required id="id_lastname">
```

В шаблоне этот же код выглядит так:

```
{% for field in form %}  
    {{ field }}  
{% endfor %}
```

Доступ к полям формы по именам

Иногда удобно вывести в шаблон поля формы не циклом, а отдельным кодом.

В шаблоне для доступа к полю применяют точечную нотацию: `form.field_name`

```
<form method="post">{% csrf_token %}  
    {{ form.firstname }}  
    {{ form.lastname }}  
    <input type="submit" value="Send message">  
</form>
```

В Python-коде доступ к полям можно получить, обратившись к объекту `form` как к словарю, где ключом является имя поля.

```
>>> print(form['firstname'])  
<input type="text" name="firstname" value="Лев" required id="id_firstname">
```

Атрибуты полей формы

При выводе формы в шаблон доступны атрибуты объекта `field`:

- **field.label** — метка поля, параметр `label` из описания поля в классе: `label="Введите имя"`
- **field.label_tag** — этот атрибут формирует полный тег `label` для поля: `<label for="id_firstname">Введите имя:</label>`
- **field.id_for_label** — здесь хранится значение, которое в HTML-теге `label` указывает, для какого именно поля формы создан этот `label`. В примере `<label for="id_firstname">Введите имя:</label>` значением тега **field.id_for_label** будет `id_firstname`
- **field.value** — значение, которое ввёл пользователь
- **field.html_name** — атрибут `name` тега `input`
- **field.help_text** — текст подсказки, который можно передать в коде
- **field.errors** — этот атрибут будет заполнен, если при проверке отправленных данных произошла ошибка

Создание нового шаблона

В документации по Bootstrap даётся такой пример HTML-кода формы:

```
<form>
  <div class="form-group">
    <label for="exampleInputEmail1">Email address</label>
    <input type="email" class="form-control" id="exampleInputEmail1" aria-
describedby="emailHelp" placeholder="Enter email">
    <small id="emailHelp" class="form-text text-muted">We'll never share
your email with anyone else.</small>
  </div>
  <div class="form-group">
    <label for="exampleInputPassword1">Password</label>
    <input type="password" class="form-control" id="exampleInputPassword1"
placeholder="Password">
  </div>
  <div class="form-group form-check">
    <input type="checkbox" class="form-check-input" id="exampleCheck1">
    <label class="form-check-label" for="exampleCheck1">Check me out</
label>
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Измените код шаблона `users/templates/signup.html` так, чтобы он соответствовал стандарту Bootstrap.

```

{% extends "base.html" %}
{% block title %}Зарегистрироваться{% endblock %}
{% block content %}

<div class="row justify-content-center">
    <div class="col-md-8 p-5">
        <div class="card">
            <div class="card-header">Зарегистрироваться</div>
            <div class="card-body">

                {% for error in form.errors %}
                    <div class="alert alert-danger" role="alert">
                        {{ error }}
                    </div>
                {% endfor %}

                <form method="post" action="{% url 'signup' %}">
                    {% csrf_token %}

                    {% for field in form %}
                        <div class="form-group row" aria-required="{% if
field.field.required %}"true"{% else %}"false"{% endif %}>
                            <label for="{{ field.id_for_label }}" class="col-md-4 col-form-label text-md-right">{{ field.label }}{% if
field.field.required %}<span class="required">*</span>{% endif %}</label>
                            <div class="col-md-6">
                                {{ field }}
                                {% if field.help_text %}
                                    <small id="{{ field.id_for_label }}-help"
class="form-text text-muted">{{ field.help_text|safe }}</small>
                                {% endif %}
                            </div>
                        </div>
                    {% endfor %}

                    <div class="col-md-6 offset-md-4">
                        <button type="submit" class="btn btn-primary">
                            Зарегистрироваться
                        </button>
                    </div>
                </form>
            </div> <!-- card body -->
        </div> <!-- card -->
    </div> <!-- col -->
</div> <!-- row -->

{% endblock %}

```

Форма регистрации теперь выглядит гораздо лучше:

Yatube

127.0.0.1:8000/auth/signup/

Войти | Регистрация

Зарегистрироваться

Имя

Фамилия

Имя пользователя*
Обязательное поле. Не более 150 символов.
Только буквы, цифры и символы @/./+/-_.

Адрес электронной почты

Пароль*
• Ваш пароль не должен совпадать с
вашим именем или другой
персональной информацией или быть
слишком похожим на неё.
• Ваш пароль должен содержать как
минимум 8 символов.
• Ваш пароль не может быть одним из
широко распространённых паролей.
• Ваш пароль не может состоять только
из цифр.

Подтверждение пароля*
Для подтверждения введите, пожалуйста,
пароль ещё раз.

Зарегистрироваться

25_Создание фильтра

В прошлом уроке мы почти сделали форму регистрации. Но наш код не выводит HTML-атрибута *class* в тегах *input*. Вот наш код:

```
<div class="form-group row" aria-required="false">
    <label for="id_first_name" class="col-md-4 col-form-label text-md-right">Имя</label>
        <div class="col-md-6">
            <input type="text" name="first_name" maxlength="30" id="id_first_name">
        </div>
</div>
```

А вот код, который должен быть:

```
<div class="form-group row" aria-required="false">
    <label for="id_first_name" class="col-md-4 col-form-label text-md-right">Имя</label>
        <div class="col-md-6">
            <input type="text" class="form-control" name="first_name" maxlength="30" id="id_first_name">
        </div>
</div>
```

Без атрибута *class* фронтендеры не смогут правильно оформить отображение формы на странице. Надо им помочь.

В этом нам поможет система фильтров в шаблонах, вы уже работали с ней. Создадим собственный фильтр для Django-шаблонов.

Создайте папку `users/templatetags`, а в ней — два пустых файла: `__init__.py` и `user_filters.py`.

У вас должна получится такая структура:

```
users
└── __init__.py
└── admin.py
└── apps.py
└── migrations
    └── __init__.py
└── forms.py
└── models.py
└── templates
    └── signup.html
└── templatetags
    ├── __init__.py
    └── user_filters.py
└── tests.py
└── urls.py
└── views.py
```

Файл `__init__.py` сообщает системе, что директория `templatetags` — это пакет, который можно импортировать в код. А в файле `user_filters.py` мы сейчас напишем код фильтра.

Фильтр даст нам возможность указывать CSS-класс в HTML-коде любого поля формы.

У объекта `field` есть метод `as_widget()`. Ему можно передать параметр с перечнем HTML-атрибутов, которые мы хотим изменить.

Финальный код фильтра будет выглядеть так:

```
from django import template
# В template.Library зарегистрированы все теги и фильтры шаблонов
# добавляем к ним и наш фильтр
register = template.Library()
```

```

@register.filter
def addclass(field, css):
    return field.as_widget(attrs={"class": css})

# синтаксис @register... , под которой описан класс addclass() –
# это применение "декораторов", функций, обрабатывающих функции
# мы скоро про них расскажем. Не бойтесь соб@к

```

Теперь в коде шаблона можно указать фильтр `addclass` с параметром `form-control`: `{{ field|addclass:"form-control" }}`.

Чтобы фильтр был доступен в шаблоне — предварительно загрузите его в шаблон тегом `{% load user_filters %}`.

Финальный код шаблона формы регистрации будет выглядеть так:

```

{% extends "base.html" %}
{% block title %}Зарегистрироваться{% endblock %}
{% block content %}
#{ загружаем фильтр #}
{% load user_filters %}



Зарегистрироваться



{% for error in form.errors %}


{{ error|escape }}


{% endfor %}

<form method="post" action="{% url 'signup' %}>
{% csrf_token %}

        {% for field in form %}
            <div class="form-group row" aria-required={% if
field.field.required %}"true"{% else %}"false"{% endif %}>
                <label for="{{ field.id_for_label }}" class="col-md-4 col-form-label text-md-right">{{ field.label }}{% if
field.field.required %}<span class="required">*</span>{% endif %}</label>
                <div class="col-md-6">

                    #{ подключаем фильтр и указываем класс #
                    {{ field|addclass:"form-control" }}

                    {% if field.help_text %}
                        <small id="{{ field.id_for_label }}-help" class="form-text text-muted">{{ field.help_text|safe }}</small>
                    {% endif %}


```

```

        </div>
    </div>
    {% endfor %}

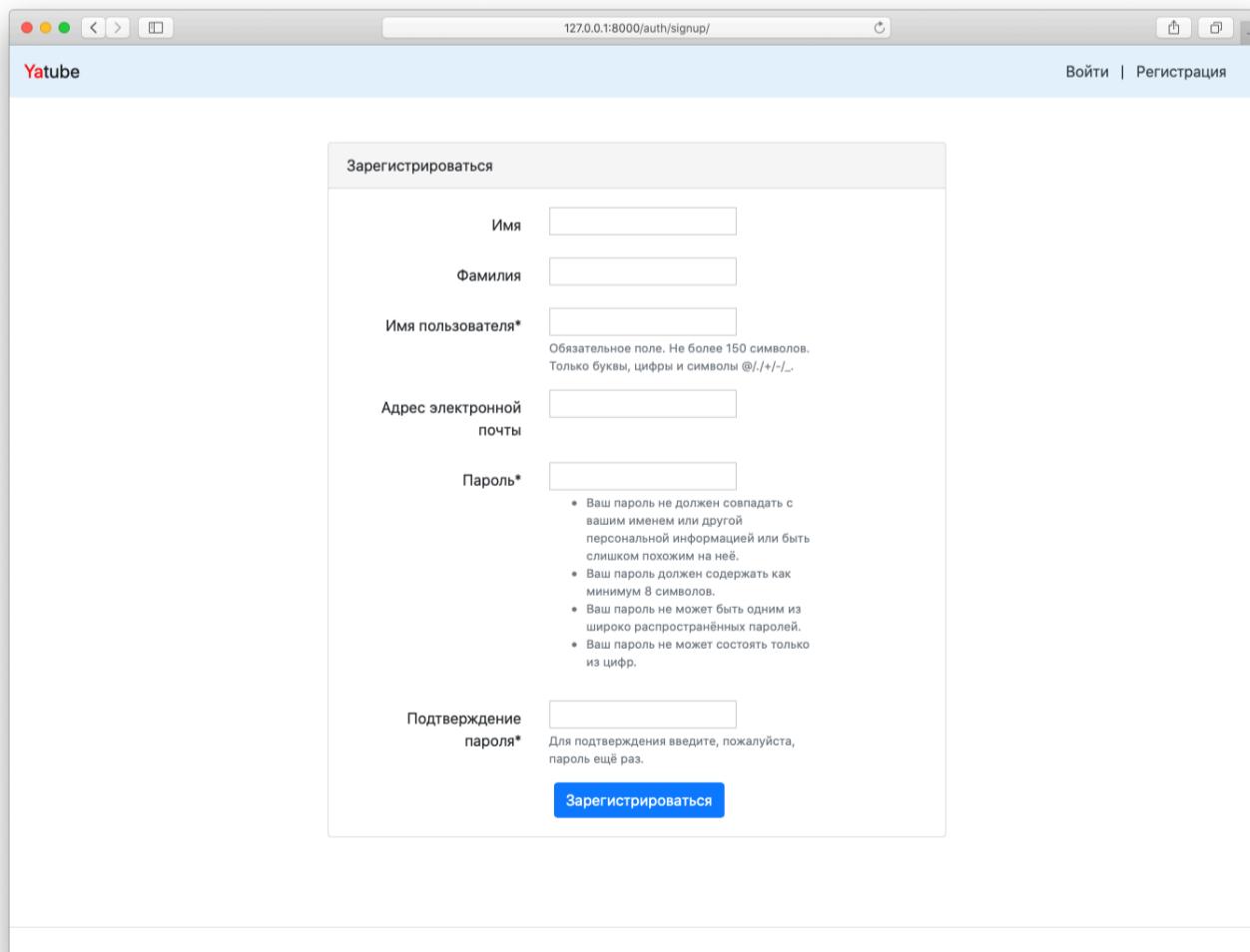
    <div class="col-md-6 offset-md-4">
        <button type="submit" class="btn btn-primary">
            Зарегистрироваться
        </button>
    </div>
</form>
</div> <!-- card body -->
</div> <!-- card -->
</div> <!-- col -->
</div> <!-- row -->

{% endblock %}

```

Внимание: обычно Django видит, что файлы проекта изменились, когда вы запускаете проект командой `$ python manage.py runserver`. Но иногда он может не заметить изменений. Перезапустите Django-проект: в окне терминала нажмите `Ctrl+C` для остановки и выполните команду `$ python manage.py runserver` для старта.

Откройте или обновите страницу регистрации. Форма выглядит чисто, красиво и профессионально:



26_Шаблоны страниц входа и восстановления доступа

Мы составили список страниц, необходимых для входа, регистрации и восстановления пароля. Этот список есть в `django/contrib/auth/views.py`:

```
class LoginView(SuccessURLAllowedHostsMixin, FormView):
    template_name = 'registration/login.html'
class LogoutView(SuccessURLAllowedHostsMixin, TemplateView):
    template_name = 'registration/logged_out.html'
class PasswordResetView(PasswordContextMixin, FormView):
    template_name = 'registration/password_reset_form.html'
class PasswordResetDoneView(PasswordContextMixin, TemplateView):
    template_name = 'registration/password_reset_done.html'
class PasswordResetConfirmView(PasswordContextMixin, FormView):
    template_name = 'registration/password_reset_confirm.html'
class PasswordResetCompleteView(PasswordContextMixin, TemplateView):
    template_name = 'registration/password_reset_complete.html'
class PasswordChangeView(PasswordContextMixin, FormView):
    template_name = 'registration/password_change_form.html'
class PasswordChangeDoneView(PasswordContextMixin, TemplateView):
    template_name = 'registration/password_change_done.html'
```

Пришло время добавить эти шаблоны в проект.

В папке `users/templates` создайте директорию `registration`, а в ней – файлы с перечисленными именами. У вас получится такая структура:

```
users
└── templates
    └── registration
        ├── logged_out.html
        ├── login.html
        ├── password_change_done.html
        ├── password_change_form.html
        ├── password_reset_confirm.html
        └── password_reset_done.html
```

```
|   |   └── password_reset_form.html  
|   └── signup.html  
├── templatetags  
│   ├── __init__.py  
│   └── user_filters.py  
├── tests.py  
└── urls.py  
└── views.py
```

Когда Django ищет запрошенный шаблон, то проходит по каждой директории `templates` в приложениях из списка `INSTALLED_APPS`, ищет там файл, чей путь и имя совпадают с `template_name`.

Давайте разберёмся с назначением этих файлов:

- **login.html** — форма входа, пользователь вводит `username` и пароль. После отправки форма проверяется, и если пользователь с таким именем и паролем найден, то он авторизуется и перенаправляется на главную страницу
- **logged_out.html** — когда пользователь переходит на эту страницу — он разлогинивается и ему показывается прощальная страница
- **password_change_form.html** — если залогиненному пользователю надо изменить пароль, то на этой странице он может указать текущий пароль и ввести новый. Если пользователь правильно ввёл текущий пароль и новый соответствует требованиям безопасности, то показывается страница **password_change_done.html**.
- **password_reset_form.html** — даёт восстановить доступ к сайту: пользователь отправляет через форму свой логин или email, а ему на почту приходит письмо со ссылкой на страницу восстановления пароля — **password_reset_confirm.html**. На этой странице пользователь указывает новый пароль, и после валидации нового пароля пользователю показывается страница **password_reset_done.html**.

login.html

Добавьте код в шаблон `users/templates/registration/login.html`

```
{% extends "base.html" %}  
{% block title %}Войти{% endblock %}  
{% block content %}
```

```
{% load user_filters %}

<div class="row justify-content-center">
  <div class="col-md-8 p-5">
    <div class="card">
      <div class="card-header">Войти на сайт</div>
      <div class="card-body">
        {% if form.errors %}
          <div class="alert alert-danger" role="alert">
            Имя пользователя и пароль не совпадают. Введите правильные данные.
          </div>
        {% endif %}

        <div class="alert alert-info" role="alert">
          Пожалуйста, авторизуйтесь.
        </div>

        <form method="post" action="{% url 'login' %}">
          {% csrf_token %}
          <input type="hidden" name="next" value="{{ next }}>
          <div class="form-group row">
            <label for="{{ form.username.id_for_label }}" class="col-md-4 col-form-label text-md-right">Имя пользователя</label>
            <div class="col-md-6">
              {{ form.username|addclass:"form-control" }}
            </div>
          </div>

          <div class="form-group row">
            <label for="{{ form.password.id_for_label }}" class="col-md-4 col-form-label text-md-right">Пароль</label>
            <div class="col-md-6">
              {{ form.password|addclass:"form-control" }}
            </div>
          </div>

          <div class="col-md-6 offset-md-4">
            <button type="submit" class="btn btn-primary">
              Войти
            </button>
            <a href="{% url 'password_reset' %}" class="btn btn-link">
              Забыли пароль?
            </a>
          </div>
        </form>
      </div> <!-- card body -->
    </div> <!-- card -->
  </div> <!-- col -->
</div> <!-- row -->

{% endblock %}
```

Страницы успешных действий: password_change_done.html, password_reset_done.html и logged_out.html

Это простые страницы для вывода сообщений о том, что операция прошла успешно.

password_change_done.html:

```
{% extends "base.html" %}  
{% block title %}Пароль изменён{% endblock %}  
  
{% block content %}  
<div class="row justify-content-center">  
  <div class="col-md-8 p-5">  
    <div class="card">  
      <div class="card-header">Пароль изменён</div>  
      <div class="card-body">  
        <p>Пароль изменён успешно</p>  
      </div> <!-- card body -->  
    </div> <!-- card -->  
  </div> <!-- col -->  
</div> <!-- row -->  
  
{% endblock %}
```

Шаблон password_reset_done.html с уведомлением об успешном восстановлении пароля создайте самостоятельно на основе шаблона password_change_done.html

Текст в password_reset_done.html поставьте такой:

- **Заголовок:** Сброс пароля прошёл успешно
- **Сообщение:** Проверьте свою почту, вам должно прийти письмо со ссылкой для восстановления пароля.

Страницу logged_out.html наполните таким текстом:

- **Заголовок:** Вы вышли из системы
- **Сообщение:** Вы вышли из своей учётной записи. Ждём вас снова!

Шаблоны изменения пароля

В файл password_change_form.html добавьте такой код:

```
{% extends "base.html" %}
```

```

{% block title %}Изменение пароля{% endblock %}
{% block content %}
{% load user_filters %}



Изменить пароль



<form method="post">
    {% csrf_token %}

    {% for field in form %}
        {# TODO: Добавьте сюда код отображения поля #}
    {% endfor %}

    <div class="col-md-6 offset-md-4">
        <button type="submit" class="btn btn-primary">
            Изменить пароль
        </button>
    </div>
</form>

    </div> <!-- card body -->
</div> <!-- card -->
</div> <!-- col -->
</div> <!-- row -->

{% endblock %}


```

Допишите этот шаблон: выведите поля формы в цикле `{% for field in form %}`

Форма сброса пароля **password_reset_form.html** очень похожа на **password_change_form.html**:

```

{% extends "base.html" %}
{% block title %}Сброс пароля{% endblock %}
{% block content %}

```

```

{% load user_filters %}

<div class="row justify-content-center">
    <div class="col-md-8 p-5">
        <div class="card">
            <div class="card-header">Чтобы сбросить старый пароль – введите
адрес электронной почты, под которым вы регистрировались</div>
            <div class="card-body">

                <form method="post">
                    {% csrf_token %}

                    {% for field in form %}
                        {# TODO: Добавьте сюда код отображения поля #}
                    {% endfor %}

                    <div class="col-md-6 offset-md-4">
                        <button type="submit" class="btn btn-primary">
                            Сбросить пароль
                        </button>
                    </div>
                </form>

            </div> <!-- card body -->
        </div> <!-- card -->
    </div> <!-- col -->
</div> <!-- row -->

{% endblock %}

```

Доработайте и этот шаблон: выведите поля формы в цикле `{% for field in form %}`

Шаблон **password_reset_confirm.html** почти полностью соответствует **password_change_form.html**, за одним исключением. Ссылка в письме может сработать только один раз. Если пользователь перейдёт по ней, ссылка устареет. Для проверки «действенности» ссылки добавим проверку переменной **validlink**. Доработайте эту заготовку шаблона:

```

{% extends "base.html" %}
{% block title %}Новый пароль{% endblock %}
{% block content %}
{% load user_filters %}

{% if validlink %}

<div class="row justify-content-center">
    <div class="col-md-8 p-5">
        <div class="card">
            <div class="card-header">Ведите новый пароль</div>
            <div class="card-body">

```

```

<form method="post">
    {% csrf_token %}

        {% for field in form %}
            #{ TODO: Добавьте сюда код отображения поля #}
        {% endfor %}

        <div class="col-md-6 offset-md-4">
            <button type="submit" class="btn btn-primary">
                Назначить новый пароль
            </button>
        </div>
    </form>

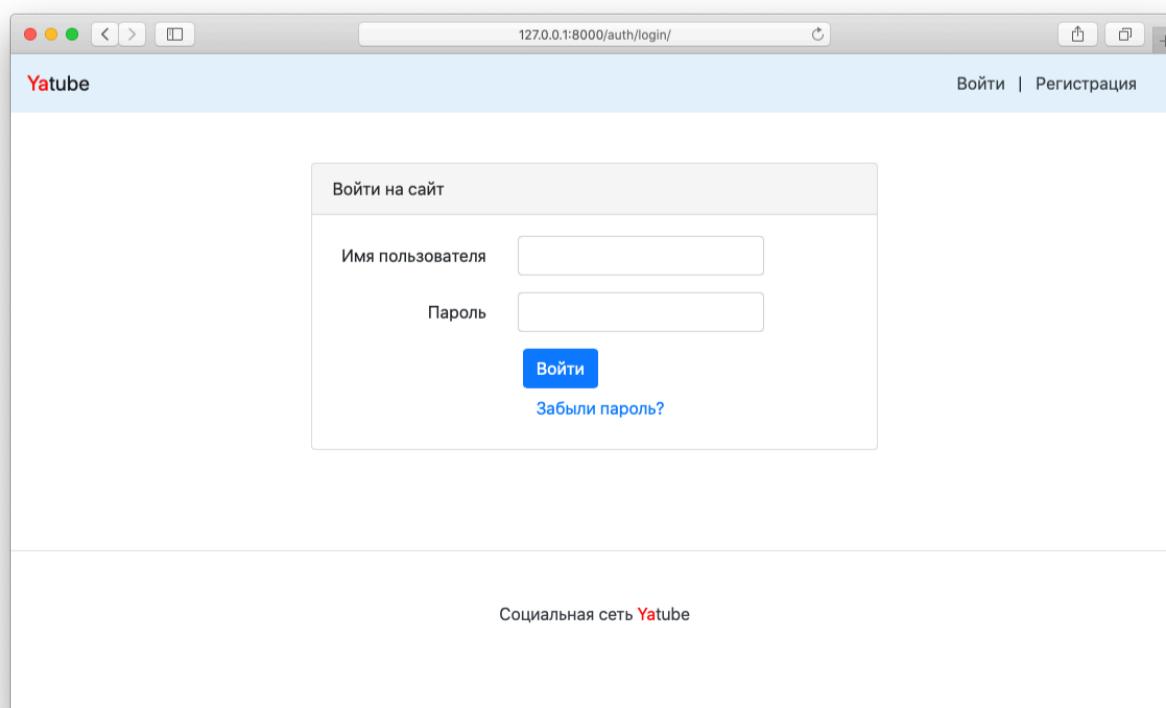
        </div> <!-- card body -->
    </div> <!-- card -->
</div> <!-- col -->
</div> <!-- row -->

{% else %}
<div class="row justify-content-center">
    <div class="col-md-8 p-5">
        <div class="card">
            <div class="card-header">Ошибка</div>
            <div class="card-body">
                <p>Ссылка сброса пароля содержит ошибку или устарела.</p>
            </div> <!-- card body -->
        </div> <!-- card -->
    </div> <!-- col -->
</div> <!-- row -->
{% endif %}

{% endblock %}

```

Проверьте свою работу, ваши новые страницы должны выглядеть приблизительно так:



127.0.0.1:8000/auth/password_reset/

Yatube Войти | Регистрация

Чтобы сбросить старый пароль введите свой адрес электронной почты под которым вы регистрировались

Адрес электронной почты*

Сбросить пароль

Социальная сеть Yatube

127.0.0.1:8000/auth/password_change/

Yatube Пользователь: admin. Изменить пароль Выйти

Изменить пароль

Старый пароль*

Новый пароль*

Подтверждение нового пароля*

Ваш пароль не должен совпадать с вашим именем или другой персональной информацией или быть слишком похожим на неё.

Ваш пароль должен содержать как минимум 8 символов.

Ваш пароль не может быть одним из широко распространённых паролей.

Ваш пароль не может состоять только из цифр.

Изменить

127.0.0.1:8000/auth/logout/

Yatube Войти | Регистрация

Вы вышли

Спасибо, что воспользовались нашим сайтом. Вы вышли из своей учетной записи. Ждем вас снова!

Социальная сеть Yatube

Финальные настройки

Осталось объяснить Django, какие страницы надо показывать пользователю после входа в аккаунт и выхода из него. Добавьте в `yatube/settings.py` такие настройки:

```
# Login

LOGIN_URL = "/auth/login/"
LOGIN_REDIRECT_URL = "index"
# LOGOUT_REDIRECT_URL = "index"
```

Значениями параметров `LOGIN_REDIRECT_URL` и `LOGOUT_REDIRECT_URL` могут быть имена URL-шаблонов из `urls.py` или обычные URL страниц.

Параметр `LOGOUT_REDIRECT_URL` заменяет адрес страницы, указанный в файле `logged_out.html`. Уберите комментарий с этой строки, если хотите перенаправлять пользователя на главную страницу после того, как он разлогинится.

27_Эмуляция почтового сервера

При развёртывании Django к проекту был автоматически подключён модуль `django.contrib.auth`. В числе прочего он содержит сценарий восстановления пароля: пользователь отправляет через специальную веб-форму свой email, и если этот адрес есть в базе, ему уходит письмо с инструкцией по восстановлению.

Но мы-то знаем, что никакое письмо никуда не уходит.

Для отправки писем в распоряжении разработчика должен быть настроенный почтовый сервер и зарегистрированный на нём аккаунт, от имени которого будет уходить письмо. Сейчас у нас всего этого нет, и при попытке восстановить пароль текст письма просто появится в консоли, но никуда не отправится и нигде не сохранится.

Можно создать свой почтовый сервер или работать с крупными почтовыми сервисами, но они могут быть недоступны, если вы работаете в корпоративной сети или, например, оказались оффлайн на одном из островов Индонезии.

Для обхода этой проблемы в Django можно настроить **эмуляцию** работы почтового сервера. Система будет производить все действия по отправке писем, но в реальности они никуда отправляться не будут. При тестировании проекта это удобно: почтовый ящик не будет завален тестовыми письмами, да и отлаживать отправку почты можно оффлайн.

Сохранение писем в файлы

В Django есть несколько модулей для отправки писем, подключить их можно через ключ конфигурации EMAIL_BACKEND в `settings.py`:

```
EMAIL_BACKEND = 'django.core.mail.backends.XXX'
```

Вместо XXX указывается название модуля.

Встроенные в Django почтовые модули могут отправлять почту по протоколу SMTP, выводить в консоль содержимое письма, сохранять письма в файлы, хранить их в памяти или просто ничего не делать (даже для «ничего не делать» написан специальный модуль).

Мы подключим модуль `filebased.EmailBackend`: он будет сохранять текст отправленных электронных писем в файлы в отдельную директорию. Добавьте в `settings.py` следующий код:

```
# подключаем движок filebased.EmailBackend
EMAIL_BACKEND = "django.core.mail.backends.filebased.EmailBackend"
# указываем директорию, в которую будут складываться файлы писем
EMAIL_FILE_PATH = os.path.join(BASE_DIR, "sent_emails")
```

Создайте папку `/sent_emails` в головной директории проекта.

Теперь при запросе на восстановление пароля система будет делать вид, что отправила письмо, но эти письма будут «отправляться» в директорию `sent_emails`.

Пройдите процедуру восстановления пароля для своего пользователя. Если вы всё настроили правильно — в `sent_emails` будут созданы файлы с текстом уведомления.

Делать не нужно, но знать полезно

Для работы с почтой в Django есть специальный модуль **Mail**, отправкой писем занимаются функции из этого модуля:

- `send_mail()` — для отправки единичного письма получателю.
- `send_mass_mail()` — для отправки множества писем.
- `mail_admins` — отправить письмо администраторам сайта. Список адресов администраторов перечисляется в списке **ADMINS** в `settings.py`.
- `mail_managers` — для отправки писем менеджерам сайта. Большой проект может управляться большим количеством сотрудников, и в списке **MANAGERS** конфига можно задать список их почтовых адресов, по которым будет идти служебная рассылка.

Отдельная функция для массовой отправки писем потребовалась потому, что каждое подключение к почтовому серверу технически обходится дорого. Надо дождаться соединения, отправить авторизационную информацию, затем отправить сообщения — при массовой рассылке эти операции необходимо оптимизировать отдельно.

Самый простой способ отправить письмо из собственной view-функции или класса — вызвать стандартную функцию Django `send_mail()` и передать ей на вход необходимые данные:

```
from django.core.mail import send_mail

send_mail(
    'Тема письма',
    'Текст письма.',
    'from@example.com', # Это поле "От кого"
    ['to@example.com'], # Это поле "Кому" (можно указать список адресов)
    fail_silently=False, # Сообщать об ошибках («молчать ли об ошибках?»)
)
```

Пока что нет необходимости делать это в проекте, но кто знает, что будет завтра.

28_Python: Декораторы

Всё, с чем мы работаем в Python — это **объекты** в терминах ООП: переменные, классы, экземпляры классов и даже импортированные модули. И функции — не исключение.

Запустите в консоли интерактивный режим Python, нам надо поэкспериментировать.

Выполните такой код:

```
def func(x, y):
    return x+y

print(type(func))
# результат: <class 'function'>
```

При создании нового объекта он сохраняется в памяти по определённому адресу, а имя добавляется в доступное пространство имен и указывает на этот адрес.

```
# посмотрим, по какому адресу в памяти сохранена функция func()
hex(id(func))
# результат: '0x21115c49280'

# встроенная функция id() показывает адрес, по которому в памяти сохранён
# объект
# а встроенная функция hex() выводит этот адрес в принятом для таких задач
# шестнадцатеричном формате
```

Создадим переменную new_func и присвоим ей значение func, точно так же, как присвоили бы переменной строковый или числовой объект:

```
new_func = func
print(new_func(1, 2))
# результат: 3
# и посмотрим, на какой адрес указывает имя new_func
hex(id(new_func))
# '0x21115c49280'
# это тот же адрес, на который указывает и имя func:
# два указателя направляют на один адрес
```

Значение первой переменной можно заменить, но вторая переменная всё равно будет указывать на прежний объект.

```
# переопределим переменную func, теперь она примет объект типа str
func = 'just a string'
print(func)
# результат: just a string

# посмотрим, на какой адрес теперь указывает имя func
hex(id(func))
# результат: '0x7ffba34096a0'
# адрес другой: был создан новый объект типа str, он сохранён по новому адресу
# и имя func указывает на него

# но переданное в new_func значение сохранилось:
# new_func всё ещё указывает на на прежний адрес, где сохранена функция
hex(id(new_func))
# результат: '0x21115c49280'
```

Старую переменную `func` мы переопределили, но `new_func` продолжает указывать на ту же самую функцию, на которую изначально указывало имя `func`.

Поскольку функция ведет себя как обычная переменная, то ее можно передавать в качестве параметра в другую функцию:

```
def apply(f, x, y):
    return f(x, y)

print(apply(new_func, 1, 2))
# результат: 3
```

Если функцию можно передать, то что мешает вернуть функцию?

```
def operation(name):

    def add(x, y):
        return x + y

    def mul(x, y):
        return x * y

    if name == 'add':
        return add

    if name == 'mul':
        return mul
```

```
# сложение
op = operation('add')
print(op(1, 3))
# результат: 4

# умножение
op2 = operation('mul')
print(op2(2, 5))
# результат: 10

# или даже так: умножение, но без промежуточной функции
print(operation('mul')(2, 5))
# результат: 10
```

Внутри функции мы создали другие функции, а потом внешним переменным присвоили ссылки на эти «внутренние» функции.

Можно сделать еще один шаг: обернуть входящую функцию нужным кодом прежде, чем её выполнить.

- Создадим функцию `wrapper()`
- Передадим ей как параметр функцию `some_func()`
- Функция `wrapper()` передаёт `some_func()` в свою внутреннюю функцию `added_value()`, которая выполняет некоторые полезные действия
- Функция `wrapper()` возвращает функцию `added_value()`

```
def wrapper(func):
    # какие-то действия с func
    _cache = {'counter': 0}
    def added_value():
        _cache['counter'] = _cache['counter'] + 1
        print("Полезная работа до начала работы функции")
        func()
        print("Полезная работа после выполнения функции")

    return added_value

def some_func():
    print("Я полезная функция")

do = wrapper(some_func)
do()
# результат:
# Полезная работа до начала работы функции
# Я полезная функция
# Полезная работа после выполнения функции
```

Можно упростить вызов, не создавая промежуточных переменных, а заменить `some_func` на саму себя, «обернуть» в функцию `wrapper()`:

```
some_func = wrapper(some_func)
some_func()
# результат:
# Полезная работа до начала работы функции
# Я полезная функция
# Полезная работа после выполнени функции
```

Теперь при каждом обращении к функции `some_func()` будет выполняться внутренняя функция `added_value()` из функции `wrapper()`

Код, размещённый в декораторе вне внутренней функции, будет выполнен лишь единожды. В декораторе `wrapper()` будет создана переменная `_cache` и значение `_cache['counter']` будет увеличиваться на единицу с каждым вызовом декоратора, но не будет создаваться каждый раз заново.

Задача по изменению работы функций достаточно распространена. Например, можно «обернуть» функцию и измерить время ее выполнения.

Или зарегистрировать функцию в реестре, чтобы потом обращаться к ней через этот реестр: мы делали так при создании странного фильтра `uglify`.

Можно одной и той же «обёрткой» настроить разные функции так, чтобы они выполнялись только один раз, сохраняли значение и при повторном обращении моментально отдавали готовый результат.

Такое решение оказалось столь популярно и удобно, что в языке была добавлена специальная конструкция: **декоратор**.

Если перед определением функции написать имя «обращающей» функции через знак @, то определяемая функция будет передаваться в «обёртку» и возвращать результат через неё.

```
@wrapper # оборачиваем some_func() в декоратор wrapper()
def some_func():
    print("Я полезная функция")
```

Упрощённый синтаксис, «синтаксический сахар», существует только для того, чтобы делать работу программиста удобнее.

Работа с аргументами

Функция-декоратор получает на вход только один параметр — декорируемую функцию, а возвращает внутреннюю функцию-исполнитель. Параметры декорируемой функции можно передать в функцию-исполнитель.

Декораторы обычно пишут универсальными: они должны принимать на вход функции с любым количеством и типом параметров. Для этого есть конструкции `*args` и `**kwargs`, это маски для получения любого количества позиционных (`*args` от *arguments*) и именованных (`**kwargs` от *keyword arguments*) аргументов. Эти конструкции могут применяться в любых функциях (не только в декораторах), имена `arg` и `kwarg` не предустановлены, но общеприняты.

В теле функции с переменной `*args` можно работать как с кортежем, а с переменной `**kwargs` — как со словарём.

```
# функция-обертка
def wrapper(func):
    # какие-то действия с func
    # функция-исполнитель
    # получает аргументы из декорируемой функции
```

```

def added_value(*args, **kwargs):
    print("Полезная работа до начала работы функции")
    func(*args, **kwargs)
    # напечатаем аргументы
    print(args)
    print(kwargs)
    print("Полезная работа после выполнени функции")
return added_value

@wrapper # декоратор
def some_func(): # декорируемая функция
    # код функции

```

Теперь любая декорируемая функция будет вызвана без ошибок.

29_Декоратор login_required

Применим декораторы на практике.

Некоторые страницы сайта должны быть доступны только залогиненным пользователям. Например, страница добавления поста, подписки на автора или форма отправки комментария. Можно делать проверку внутри каждой view-функции, приблизительно так:

```

def only_user_view(request):
    if not request.user.is_authenticated():
        return response.redirect(
            # если пользователь не авторизован – отправляем его на страницу
логина
            to_page = '/auth/login',
            # в GET-параметре передадим переменную next, в ней сохраним адрес
текущей страницы,
            # чтобы вернуть на неё пользователя после авторизации
            params = {'next': current_page}
        )
    # остальной полезный код функции

```

И такая проверка должна быть в каждой view-функции, работающей с пользовательскими страницами.

На сайте будет десяток view-функций для пользовательских страниц, в каждой из них придётся повторять этот код. Это будет загромождать проект, а при необходимости внести изменения придётся править десяток функций.

Но можно один раз написать декоратор, а потом одной строкой в коде «обращивать» в него любую view-функцию:

```
def user_only(func):
    def check_user(request, *args, **kwargs):
        # Мы знаем, что у view-функций первый аргумент всегда request
        if not request.user.is_authenticated():
            return response.redirect(
                to_page = '/auth/login',
                params = {'next': current_page})
        )
    func(request, *args, **kwargs)
    return check_user
```

Теперь нам надо лишь добавить декоратор к view-функциям:

```
from .utils import user_only

@login_required
def only_user_view(request):
    # Только полезный код!
```

В Django уже есть встроенный декоратор для таких случаев, он называется `login_required`. С его помощью можно закрыть страницы от неавторизованных пользователей.

Этот декоратор устроен несколько сложнее, чем наш: он умеет принимать параметры, работать с *Class Based Views* и различать, оборачивает ли он функцию или метод класса.

Декоратор `login_required` импортируется из `django.contrib.auth.decorators`:

```
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    ...
```

Декоратор `@login_required` и шаблон `login.html`

Измените шаблон `users/templates/registration/login.html` — добавьте блок с проверкой переменной `next`. Если незалогиненный пользователь обратится к странице проекта, доступной только для авторизованных пользователей, мы направим его на страницу авторизации. Если он авторизуется — мы вернём его на ту страницу, с которой он пришёл. Адрес этой страницы будет передан в GET-параметре в переменной `next`: `/auth/login?next=any-page-url`

Текст из блока `{% if next %}` будет отображён именно при переадресации пользователя на страницу авторизации.

```
{% extends "base.html" %}  
{% block title %}Войти{% endblock %}  
{% block content %}  
{% load user_filters %}  
  
<div class="row justify-content-center">  
    <div class="col-md-8 p-5">  
        <div class="card">  
            <div class="card-header">Войти на сайт</div>  
            <div class="card-body">  
                {% if form.errors %}  
                    <div class="alert alert-danger" role="alert">  
                        Имя пользователя и пароль не совпадают. Введите правильные данные.  
                    </div>  
                {% endif %}  
  
                {% if next %}  
                    <div class="alert alert-info" role="alert">  
                        Вы обратились к странице, доступ к которой возможен только для  
зарегистрированных пользователей.<br>  
                        Пожалуйста, авторизуйтесь.  
                    </div>  
                {% else %}  
                    <div class="alert alert-info" role="alert">  
                        Пожалуйста, авторизуйтесь.  
                    </div>  
                {% endif %}  
  
                <form method="post" action="{% url 'login' %}">  
                    {% csrf_token %}  
                    <input type="hidden" name="next" value="{{ next }}>  
                    <div class="form-group row">  
                        <label for="{{ form.username.id_for_label }}" class="col-md-4  
col-form-label text-md-right">Имя пользователя</label>  
                        <div class="col-md-6">  
                            {{ form.username|addclass:"form-control" }}  
                        </div>  
                    </div>  
  
                    <div class="form-group row">  
                        <label for="{{ form.password.id_for_label }}" class="col-md-4  
col-form-label text-md-right">Пароль</label>  
                        <div class="col-md-6">  
                            {{ form.password|addclass:"form-control" }}  
                        </div>  
                    </div>  
  
                    <div class="col-md-6 offset-md-4">  
                        <button type="submit" class="btn btn-primary">  
                            Войти  
                        </button>  
                        <a href="{% url 'password_reset' %}" class="btn btn-link">  
                            Забыли пароль?  
                        </a>  
                    </div>  
                </form>  
            </div> <!-- card body -->
```

```
</div> <!-- card -->
</div> <!-- col -->
</div> <!-- row -->

{% endblock %}
```

Теперь view-функции тех страниц, куда разрешён доступ только под логином (это, например, страница создания нового поста), можно «обернуть» декоратором `@login_required` — и автоматически будет проводиться проверка и переадресация неавторизованных пользователей на страницу логина.

30_Валидация форм

Вы уже поработали с моделями и формами, созданными на основе *Generic Views*. Чтобы увидеть полную картину — научимся обрабатывать формы во view-функциях.

В самом общем случае работа с формами происходит в таком порядке:

- Разработчик создал модель, на её основе создал форму, эту форму вывел в шаблон. Можно создать форму и без модели, если хранить полученные данные не нужно.
- Пользователь заходит на страницу с формой, заполняет её и нажимает «Отправить», браузер пользователя отправляет POST-запрос с данными на URL, указанный в параметре `action` формы.
- На сервере данные из формы проверяются и обрабатываются во view-функции.
- Если данные успешно прошли проверку — они передаются для дальнейшей обработки и сохранения в базе, а пользователь перенаправляется на страницу с информацией об успешной отправке формы.
- Если отправленные данные не прошли проверку, пользователю отправляется уведомление об ошибке.

В Django есть полная инфраструктура для работы с формами:

- Формы Django можно связывать с объектами модели.
- На основе моделей можно быстро, буквально несколькими строками кода, создавать новые формы.

- Данные, полученные от пользователя, автоматически разбираются обработчиком запроса.
- Django умеет передавать данные, полученные от пользователя, в объект формы.
- У форм есть система валидации — проверки качества переданных данных. Пользователь может ошибиться или раньше времени нажать кнопку «Отправить» — в этих случаях данные не пройдут проверку и пользователь получит сообщение об этом.
- Во все формы Django по умолчанию встроена защита от злонамеренной отправки данных из другого источника, *Cross Site Request Forgery* (CSRF).
- В Django предусмотрен набор встроенных виджетов — шаблонов, формирующих HTML-код формы: поля для ввода текста (HTML-тег `<input type="text">`), поля многострочного ввода `<textarea>`, чекбоксы `<input type="checkbox">`, выпадающие списки `<select>` и множество других, стандартных и нестандартных, элементов.

Проще всего создать форму на основе существующей модели: достаточно написать класс формы, ссылающийся на модель. Django автоматически сопоставит имена и типы полей и подберёт стандартные виджеты для них:

```
from django.forms import ModelForm
from post.models import Post

# создание класса формы
class PostForm(ModelForm):
    class Meta:
        model = Post
        fields = ['pub_date', 'text']

# создание формы для отправки постов
form = PostForm()

# создание формы для редактирования конкретного объекта модели Post
# запрашиваем объект
post = Post.objects.get(pk=3)
# передаём запрошенный объект в форму
form = PostForm(instance=post)
```

Если форма привязана к определённому объекту модели, то в неё будут выведены данные этого объекта. В нашем примере в форму будет выведен пост с `pk=3`, его можно будет изменить и сохранить изменения.

Для определённых типов полей модели Django автоматически подберёт особые виджеты формы:

- если в модели есть поле типа *ForeignKey*, то в форме будет отрисовано поле выбора *ModelChoiceField*, `<select>` для выбора объекта связанной модели;
- для поля модели *ManyToManyField* в форме будет применён виджет *ModelMultipleChoiceField*, поле для множественного выбора из списка;
- для поля модели *SlugField* в форме отобразится поле *SlugField*. Это специальное текстовое поле `<input type="text">` умеет создавать красивые URL для объектов и проверять, что строка состоит только из разрешённых для URL символов. Например, для этого урока *SlugField* предложил бы URL `/validatsiya-form`

Валидация форм

Данные, переданные из форм, проверяются функциями-валидаторами. Валидатор обращается к данным (или получает их в качестве аргумента) и проверяет их значение. В случае ошибки вызывается исключение `ValidationError`. В Django есть множество встроенных валидаторов, но можно написать и собственный.

Валидация поля формы может быть многоэтапной. Значения, полученные после валидации каждого из полей, доступны в словаре `form.cleaned_data`. Ключи в этом словаре — названия полей формы.

Создадим форму для отправки сообщений администратору сайта:

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
```

Валидация данных устроена так:

- Для любого поля модели или формы можно указать аргумент `validators` и в нём перечислить функции-валидаторы для этого поля. При получении данных валидаторы вызываются автоматически, им передаётся значение поля, и в случае ошибки валидации вызывается исключение `ValidationError`

```

# функция-валидатор:
def validate_positive(value):
    if value <= 0:
        raise forms.ValidationError(
            'Значение меньше нуля',
            params={'value': value},
        )

# можно указать валидатор для свойства (поля) модели
class MyModel(models.Model):
    number_field = models.IntegerField(validators=[validate_positive])

# ... а можно указать валидатор для свойства (поля) формы
class MyForm(forms.Form):
    number_field = forms.IntegerField(validators=[validate_positive])

```

- В классе формы для каждого отдельного поля можно создать метод `clean_<имя поля>`, он вызовется автоматически во время валидации данных. Этот метод не получает никаких дополнительных параметров: он должен самостоятельно запросить значение из словаря `cleaned_data`. Такой протокол создан для того, чтобы можно было совместить работу всех валидаторов. Значение, которое вернёт этот метод, обновит значение соответствующего элемента словаря `cleaned_data`.

Напишем валидатор, который заблокирует отправку формы, если в ней нет благодарности в адрес администратора сайта. Зачем нужно письмо, если в нём тебе не говорят «спасибо»?

```

class ContactForm(forms.Form):
    # форма обратной связи
    ...
    # метод-валидатор для поля subject
    def clean_subject(self):
        data = self.cleaned_data['subject']
        # если пользователь не поблагодарил администратора – считаем это ошибкой
        if "спасибо" not in data.lower():
            raise forms.ValidationError("Вы обязательно должны нас поблагодарить!")

        # метод-валидатор обязательно должен вернуть очищенные данные, даже если не изменил их
        return data

```

Классам полей формы и модели тоже можно присвоить методы валидации. Например, вы можете создать поле для ввода цвета `ColorField` с валидацией — и валидация этого поля будет работать во всех формах, где оно применяется.

Форма считается валидной, если метод формы `.is_valid()` возвращает `True`.

После валидации все данные будут сохранены в `form.cleaned_data`, и дальнейшая работа с информацией из формы идёт именно с этими данными:

```
if form.is_valid():
    subject = form.cleaned_data['subject']
    message = form.cleaned_data['message']
    sender = form.cleaned_data['sender']
    cc_myself = form.cleaned_data['cc_myself']
```

Работа с пользовательскими данными

Создаём класс формы:

файл forms.py

```
from django import forms

# создаём форму
class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
```

Данные, отправленные клиентом на сервер, доступны в словаре `request.POST`. Создадим объект формы и передадим в него полученную информацию:

файл views.py

```
from django.shortcuts import redirect

def user_contact(request):
    # проверим, пришёл ли к нам POST-запрос или какой-то другой:
    if request.method == 'POST':
        # создаём объект формы класса ContactForm и передаём в него полученные
        # данные
        form = ContactForm(request.POST)
        # проверяем данные на валидность:
        # ... здесь код валидации ...

        if form.is_valid():
            # обрабатываем данные формы, используя значения словаря
            form.cleaned_data
            # возвращаем результат
```

```

# Функция redirect перенаправляет пользователя
# на другую страницу сайта, чтобы защититься
# от повторного заполнения формы, если посетитель
# сайта случайно перезагрузит страницу
return redirect('/thank-you/')

# если не сработало условие if form.is_valid() и данные не прошли
валидацию
# сработает следующий блок кода,
# иначе команда return прервала бы дальнейшее исполнение функции

# вернём пользователю страницу с HTML-формой и передадим полученный
объект формы на страницу,
# чтобы вернуть информацию об ошибке

# заодно автоматически заполним прошедшими валидацию данными все поля,
# чтобы не заставлять пользователя второй раз заполнять их
return render(request, 'contact.html', {'form': form})

form = ContactForm()
return render(request, 'contact.html', {'form': form})

```

31_Что такое тестирование

Каждая новая задача затрагивает код сразу нескольких файлов. С ростом сложности проекта любое изменение начнёт порождать волны правок в функциях, шаблонах, потребует создания файлов миграции и других изменений в проекте. Самый непреодолимый предел, с которым сталкивается программист — это предел возможностей его памяти и внимания.

В прошлом спринте вы делали форму по обмену дисками. В форме были поля «Исполнитель» и «Название альбома», но не было поля «Год выпуска». Представьте себе, что перед сдачей проекта коллекционер просит вас добавить это поле. Вот какие фрагменты кода придётся исправить:

- Объект формы, в котором нет необходимого поля;
- View-функцию index, которая вызывает функцию send_msg;
- Определение самой функции send_msg(): ей надо будет передавать дополнительный параметр date;
- Шаблон текста письма: в него надо добавить новое поле;
- Возможно, изменений потребует и шаблон формы в файле index.html;

- А если бы данные из формы сохранялись в базу, то пришлось бы править модель и делать миграцию.

И всё это надо помнить и исправлять в крошечном проекте с одной-единственной формой!

Чем больше проект, тем сложнее вносить изменения. А если над проектом трудится коллектив, то программист вполне может и не знать, что его код использует кто-то ещё; небольшое исправление приведет к тому, что часть проекта просто перестанет работать.

Виды тестирования

Вокруг проверки качества проектов возникла огромная инфраструктура и отдельная культура. Вы слышали о тестировщиках и, возможно, сами занимались тестированием. Тестировщики пытаются проиграть все возможные сценарии и найти ошибки в работе проекта. Ручная проверка называется **мануальное тестирование**. Есть особая культура мануального тестирования: иногда тестировщик просто эмулирует поведение пользователя, но чаще у него есть конкретные сценарии, которые он реализует в интерфейсе проекта. Простейший сценарий тестирования формы входа на сайт:

- Пользователь вводит правильный логин и пароль иходит на сайт;
- Пользователь вводит неправильный логин и пароль, получает отказ и видит приглашение восстановить пароль;
- Пользователь вводит правильный email в форму восстановления доступа и получает письмо со ссылкой на форму изменения пароля;
- Пользователь вводит неправильный email в форму восстановления доступа и получает сообщение «такой email не найден».

Тестировщик проходит по таким сценариям и проверяет работу системы. По мере развития в проекте появляются всё новые и новые сценарии. Прежде чем сдать свою работу, программист обращается к тестировщику для проверки сценариев и функциональности.

После внесения изменений проводят и **регRESSIONное тестирование** — проверку уже работающих частей системы на ошибки, которые могли появиться в связи с добавлением новой функциональности.

Изменение одной части проекта может вызвать ошибки («регрессии») в другой части. Например, если различные view-функции отправляют сообщения через функцию `send_msg()`, а программист изменит её, это может привести к тому, что часть view-функций перестанет работать.

Мануальное тестирование — это долго и утомительно, и эту работу автоматизировали. **Автоматическое тестирование** может работать на разном уровне: эмулировать вызов функций и методов или даже имитировать работу пользователя.

В этом спринте тесты будут проверять работу нашего кода. Продолжая наш пример, можно написать программу, которая протестирует функцию `send_msg()`. А можно написать тест, который будет проверять работу view-функции.

Тесты для небольших частей кода называются **юнит-тестами**, а тесты, проверяющие работу внешних программных интерфейсов — это **интеграционные тесты** (они обеспечивают интеграцию между разными системами). Те и другие — **функциональные тесты**, поскольку испытывают они функциональность систем. Интеграционные тесты проверяют реальные задачи, например, может ли пользователь добавить запись на сайт. Юнит-тест проверяет работу участков кода, например — правильно ли функция обрабатывает входящие параметры.

Есть много иных видов тестирования. Можно тестировать производительность или предельную нагрузку на систему. А можно проверять систему на эстетическую красоту или удобство работы пользователей с интерфейсами. Есть даже тестирование, проверяющее объем выбросов CO₂ на единицу операции.

Когда надо писать тесты

Написание тестов — это большая часть работы программистов. Иногда даже вначале пишутся тесты, а только потом пишется код, соответствующий этим тестам. Эта практика называется **Test Driven Development (TDD)**.

Соотношение числа протестированных строк кода к общему количеству значимых строк называется **процентом покрытия тестами**. Есть специальные утилиты, позволяющие отследить, была ли задействована конкретная строка (или её часть) во время запуска тестов.

Идеальное (и часто практически недостижимое) покрытие кода — стопроцентное, когда написаны тесты для всех позитивных и негативных исходов, для каждой функции и каждого из состояний классов и объектов, которые они могут порождать. А это значит, что объём тестов может превышать объём кода проекта!

Традиционно принимается на веру, что покрытый тестами код содержит меньше ошибок и доставит меньше проблем при внесении изменений. При этом так же принимается, что покрытый тестами код всё равно может содержать ошибки. Реальность такова, что требования к одному и тому же коду могут меняться, так же как может меняться среда и условия его выполнения.

Программисты очень часто сталкиваются с преодолением неучтённых ситуаций:

- База данных прекрасно справляется с миллионами записей в таблице. Но если в таблицу добавляются миллионы записей в секунду, например, для подсчёта объёма потребленного трафика сотового оператора — возникнет ошибка.
- Библиотека для обработки изображений, на 100% покрытая тестами, получит для обработки изображения со спутника с таким разрешением, о существовании которого создатели библиотеки даже не подозревали — и зависнет.
- Генератор случайных чисел прекрасно справляется с симуляцией подбрасывания кубика в компьютерной игре — но оказывается предсказуем и приводит к деградации стойкости криptoалгоритмов.

Код пишут люди, а людям свойственно ошибаться. Мы часто считаем, что «всё сделали правильно» до тех пор, пока не отойдём на шаг от своей работы и не взглянем на неё свежим взглядом. Тесты помогают посмотреть на код со стороны.

Когда надо писать тесты и сколько их должно быть? Это вопрос одновременно и философский, и прагматичный.

Тесты — это тоже код, работа, которая стоит денег и времени. Помимо стоимости часа работы есть ещё цена времени для вывода продукта на рынок. Но не проверять и не тестировать свою работу — это значит постоянно сталкиваться с проблемами и исправлять ошибки в ходе

эксплуатации, что всегда дороже. Ошибки могут привести к потере клиента или даже к закрытию бизнеса.

Если на проекта **Poemnotes** возникнет ошибка, не позволяющая пользователю зарегистрироваться, и при этом идёт реклама, приводящая на сайт новых клиентов, то все деньги рекламной компании будут потрачены впустую.

Поэтому давайте делать свою работу и разберёмся, как покрыть код проекта тестами!

32_Запуск первых тестов

В мире Python есть несколько популярных фреймворков тестирования. В стандартную библиотеку языка входит фреймворк **unittest**, он принят в Django как основа для написания тестов. Помимо него есть и другие библиотеки, но мы будем тестировать код стандартными инструментами.

При создании нового приложения в его директории появился файл `tests.py`. Он пуст, но уже включен в инфраструктуру тестирования проекта.

Давайте запустим все тесты проекта (спойлер: их нет).

```
(venv) $ python manage.py test  
System check identified no issues (0 silenced).
```

```
-----  
Ran 0 tests in 0.000s  
  
OK  
# нет тестов – нет и ошибок!
```

Тестирование в Django организовано достаточно удобно и гибко. При выполнении команды общего запуска тестов система ищет файл `tests.py` в каждом приложении и запускает тесты в этом файле. Если в проекте много приложений, тесты запускаются по очереди.

Пора написать первый тест. Добавьте в файл `posts/tests.py` такой код:

```
from django.test import TestCase  
  
# Create your tests here.  
  
class TestStringMethods(TestCase):
```

```
def test_length(self):
    self.assertEqual(len('yatube'), 6)
```

В командной строке запустите выполнение тестов:

```
(venv) $ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
. # внимание, эта точка в выводе означает, что запущенный тест пройден
-----
Ran 1 test in 0.001s
```

OK

```
Destroying test database for alias 'default'...
```

Система обнаружила и запустила тест, он выполнился успешно.

Точка в выводе означает, что тест пройден успешно. При провале теста будет выведен символ F (*Failed*).

В больших проектах могут быть сотни или даже тысячи тестов, их совместный запуск отнимает много времени — до нескольких десятков минут. Если вы работаете над конкретной задачей, то не касающиеся её тесты отнимут время и не принесут пользы. Так что есть смысл запускать только часть тестов, указывая их адреса:

```
# Запускаем все тесты приложения posts
(venv) $ python manage.py test posts
...skip...
```

```
# Запускаем все тесты приложения posts которые расположены в файле tests
(venv) $ python manage.py test posts.tests
...skip...
```

```
# Запускаем конкретный класс unit-тестов
(venv) $ python manage.py test posts.tests.TestStringMethods
...skip...
```

```
# Запускаем метод test_length() из класса TestStringMethods
# из файла tests.py из директории posts
(venv) $ python manage.py test posts.tests.TestStringMethods.test_length
...skip...
```

Дополнительные параметры запуска можно узнать, выполнив команду `python manage.py test -h`.

Терминология тестирования

Файл `tests.py` может содержать множество классов с методами-тестами. Такой файл называется «набором тестов» (на английском — *test suite*).

Каждый отдельный метод обычно тестирует работу какого-то одного логического элемента, фрагмента кода. По-английски такой кусочек кода называется *test case*, а на русском обычно хватает слова «тест».

Для тестирования бывают необходимы начальные данные, например — записи в базе или какой-то файл с исходными данными. Такие данные по-английски называются *test fixtures*; возможно, кто-нибудь и перевёл это выражение удачно, но в повседневной жизни это называют *фиксатурами*.

Часть фреймворка тестирования, отвечающая за подготовку окружения и запуск тестов, по-английски называется *test runner*, но в русском языке эту часть обычно отдельно не выделяют.

Базовые предположения

Код тестов выглядит самобытно и непривычно, он не похож на знакомый вам код: вместо обычной логики тесты состоят из предположений (*assertion*), которые в ходе теста подтверждаются (в этом случае тест пройден) или опровергаются (тест провален).

Вернемся к тесту, который мы запускали:

```
# Каждый логический набор тестов – это класс,
# который наследуется от базового класса TestCase
from django.test import TestCase

# Каждый класс – это набор тестов. Имя такого класса принято начинать со слова
# Test.
# В файле может быть множество наборов тестов,
# не обязательно иметь один класс для всего приложения.
class TestStringMethods(TestCase):

    # Каждый отдельный метод в наборе тестов должен начинаться со слова
    test
        # таких методов-тестов в наборе может быть множество.
    def test_length(self):
        # В этой строке находится собственно тест который проверяет
        # предположение (assertion) являются ли переданные параметры
        # эквивалентными (equal)
        self.assertEqual(len('yatube'), 6)
```

Выражение `self.assertEqual(len('yatube'), 6)` — ключевая строка теста, она проверяет предположение, что значение первого параметра эквивалентно второму параметру. Класс `TestStringMethods` унаследован от класса `TestCase` — это базовый класс для тестов в Django, он расширяет работу стандартного класса `unittest.TestCase`, добавляя к нему дополнительный набор предположений.

Вот доступный список простых методов-предположений:

- `assertEqual(a, b)`, проверка на эквивалентность. Проверяет, что `a == b`
- `assertNotEqual(a, b)`, проверка на неравенство. То же что и `a != b`
- `assertTrue(x)`, проверка на истину, `bool(x) is True`
- `assertFalse(x)`, проверка на ложность, `bool(x) is False`
- `assertRaises()`, проверка, что метод порождает исключение
- `assertIs(a, b)`, проверка на тождественность, `a is b`
- `assert IsNot(a, b)`, проверка на нетождественность, `a is not b`
- `assertIsNone(x)`, проверка на тождественность `None`, `x is None`
- `assertIsNotNone(x)`, проверка на нетождественность `None`, `x is not None`
- `assertIn(a, b)`, проверка на вхождение в множество, `a in b`
- `assertNotIn(a, b)`, проверка на невхождение в множество, `a not in b`
- `assertIsInstance(a, b)`: является ли `a` экземпляром класса `b`, `isinstance(a, b)`
- `assertNotIsInstance(a, b)`: не является ли `a` экземпляром класса `b`, `not isinstance(a, b)`

Каждому из этих методов можно передать параметр `msg`, он делает вывод более информативным. Измените набор тестов в файле `tests.py` так:

```
from django.test import TestCase

class TestStringMethods(TestCase):
    def test_length(self):
        self.assertEqual(len('yatube'), 6)

    def test_show_msg(self):
        # действительно ли первый аргумент – True?
        self.assertTrue(False, msg="Важная проверка на истинность")
```

И выполните его:

```
(venv) $ python manage.py test posts
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.F # первый тест пройден (точка), второй тест провален (F)
=====
```

```
FAIL: test_show_msg (posts.tests.TestStringMethods)
-----
Traceback (most recent call last):
File "/Dev/Yatube/yatube/posts/tests.py", line 11, in test_show_msg
    self.assertTrue(False, msg="Важная проверка на истинность")
AssertionError: False is not true : Важная проверка на истинность
# Нет, False – это не True!
-----
Ran 2 tests in 0.001s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Предположение оказалось ложным, тест провален, а мы получили явный и читаемый сигнал о том, что ожидалось в этом тесте.

Расширенный набор предположений

Django расширяет список базовых assert-методов, добавляя специфические для web-разработки методы тестирования форм, ответов view-функций и классов.

В работе нам пригодятся такие методы:

- `assertRedirects(response, expected_url, status_code=302, target_status_code=200, msg_prefix='', fetch_redirect_response=True)`: проверка предположения, что ответ view-функции содержит редирект на нужный адрес, заодно можно проверить HTTP-код ответа страницы и код ответа адреса, на который ожидается редирект.
- `assertURLEqual(url1, url2, msg_prefix='')`: проверка предположения, что два адреса эквивалентны. Например, `/path/?x=1&y=2` — то же самое, что и `/path/?y=2&x=1`.
- `assertContains(response, text, count=None, status_code=200, msg_prefix='', html=False)`: содержит ли ответ искомый текст, дополнительно можно проверить количество вхождений.
- `assertNotContains(response, text, status_code=200, msg_prefix='', html=False)`: проверка предположения, что искомого текста нет в ответе view-функции.
- `assertFormError(response, form, field, errors, msg_prefix='')`: проверка на ошибки при валидации формы.

- `assertTemplateUsed(response=None, template_name=None, msg_prefix='', count=None)`: проверка предположения, что определенный шаблон был применён для формирования ответа.
- `assertTemplateNotUsed(response=None, template_name=None, msg_prefix='')`: проверка предположения, что определенный шаблон не использовался для формирования ответа.
- `assertHTMLEqual(html1, html2, msg=None)`: проверка предположения, что оба переданных HTML-документа одинаковы. При этом оба переданных HTML очищаются от пробельных символов, а порядок следования атрибутов в тегах не считается отличием.
- `assertHTMLNotEqual(html1, html2, msg=None)`: операция, обратная предыдущей: два HTML-документа сравниваются на неравенство с учетом таких же условий, как в предыдущем методе.

Расширенные методы работают достаточно интеллектуально. Вот пример из документации метода `assertHTMLEqual`: оба предложенных варианта сравнения не вызовут ошибки, тест будет пройден, хотя, на первый взгляд, сравниваемые фрагменты кода заметно отличаются.

```
self.assertHTMLEqual(
    '<p>Hello <b>&#x27;world&#x27;!</p>',
    '''<p>
        Hello <b>\'world\'! </b>
    </p>'''
)
self.assertHTMLEqual(
    '<input type="checkbox" checked="checked" id="id_accept_terms" />',
    '<input id="id_accept_terms" type="checkbox" checked>'
)
```

33_Подготовка условий для запуска тестов

Перед выполнением тестов иногда нужно подготовить определённые условия для их запуска: создать пользователя с определённым уровнем доступа или эмулировать работу браузера, чтобы хранить информацию о пользовательской сессии.

Подготовка контекста применяется в различных системах тестирования и традиционно называется *tear up* или *setup*, а функции, выполняющиеся после теста, обычно называют *tear down* или *cleanup*.

В Django и в *unittest* функция, выполняющаяся до начала тестов, называется `setUp()`, а выполняющаяся после окончания — `tearDown()`.

В примере используется `unittest.TestCase`; для Django код тот же.

```
# file: test_simple.py
from unittest import TestCase

class SimpleTest(TestCase):
    def setUp(self):
        print("SetUp")

    def test_one(self):
        print("One")
        self.assertEqual(int("1"), 1)

    def test_two(self):
        print("Two")
        self.assertTrue("2")

    def tearDown(self):
        print("tearDown")
```

Этот код можно запустить без Django. Команды `print()` ломают стандартный вывод, но зато позволяют понять порядок выполнения методов. Перед вызовом каждого теста запускается метод `setUp()`, после выполнения — `tearDown()`.

```
$ python -m unittest test_simple
SetUp
One
tearDown
 SetUp
Two
tearDown
.
-----
Ran 2 tests in 0.000s
```

OK

Создание тестового веб-клиента

При тестировании Django-проектов не обойтись без веб-клиента, эмулятора браузера, который будет обращаться к страницам сайта, отправляя GET- и POST-запросы, и принимать ответы сервера.

Если ответ содержит редирект, клиент сможет его отследить и считать ответ страницы, на которую осуществляется переход. Главное отличие тестового клиента от обычного браузера в том, что он имеет доступ «под капот» Django и знает, какие переменные и шаблоны были задействованы при рендеринге страницы.

В настройках проекта `poemnotes/settings.py` измените значение ключа `ALLOWED_HOSTS`:

```
ALLOWED_HOSTS = [  
    "localhost",  
    "127.0.0.1",  
    "[::1]",  
    "testserver",  
]
```

Это перечень адресов, с которых серверу разрешено принимать запросы. В этом списке есть и специальный адрес тестового сервера.

Теперь в интерактивном режиме можно эмулировать работу с сайтом через тестовый клиент. Надо только его создать. Для этого в модуле `django.test` есть класс `Client`: каждый экземпляр этого класса — это «как-бы-браузер», которым можно управлять из кода.

Примеры из следующего листинга **не сработают в вашем проекте**: скорее всего, у вас не зарегистрирован пользователь с логином `terminator`.

Проверить работу этого кода можно

- заменив в коде логин и пароль на данные любого зарегистрированного в вашем проекте пользователя
- или предварительно создав пользователя с логином `terminator` и паролем `skynetMyLove`.

```
(venv) $ python manage.py shell  
Python 3.8.0 (default, Nov 22 2019, 23:37:58)  
[Clang 11.0.0 (clang-1100.0.33.12)] on darwin
```

```
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from django.test import Client

# создаём клиент, эмулятор веб-браузера
>>> c = Client()

# – Браузер, сделай POST–запрос к странице логина
# и передай значения переменных username и password!
>>> response = c.post('/auth/login/', {'username': 'terminator', 'password': 'skynetMyLove'})

# какой код вернула страница при запросе?
>>> response.status_code
302
# после авторизации Django переадресует пользователя,
# но в клиенте–эмодуляторе по умолчанию переадресация запрещена, потому вернулся
# статус 302

# разрешим переадресацию: follow=True
>>> response = c.post('/auth/login/', {'username': 'terminator', 'password': 'skynetMyLove'}, follow=True)
>>> response.status_code
200 # Терминатор успешно залогинился, страница вернула код 200
```

Возможности тестового клиента

Тестовый клиент может делать стандартные HTTP-запросы всеми возможными методами (GET, POST, DELETE и пр.), получать информацию о шаблонах и о словаре context, переданному в шаблон при рендеринге, создавать и авторизовать пользователей.

Во время тестов вся работа идёт с временной копией базы, после тестирования основная база остаётся без изменений, а временная база удаляется из памяти.

client.get()

Метод .get() делает запрос к странице path и передаёт необходимые параметры: get(path, data=None, follow=False, secure=False, **extra).

```
>>> c = Client()
>>> c.get('/search/', {'query': 'утро', 'page': 7})
```

Такое обращение эквивалентно обращению к странице /search/?query=утро&page=7. Параметр follow разрешает клиенту совершить переход, если ответ содержит редирект:

```
>>> response = c.get('/go/', follow=True)
>>> response.redirect_chain # показать адреса редиректа
[('http://testserver/step1/', 302), ('http://testserver/end/', 302)]
```

client.post()

Для отправки данных методом POST у эмулятора клиента есть метод `post()`: `post(path, data=None, content_type=MULTIPART_CONTENT, follow=False, secure=False, **extra)`.

```
>>> c = Client()
>>> c.post('/login/', {'name': 'terminator', 'passwd': 'skynetMyLove'})
```

Помните, что форма логина в Django после авторизации редиректит пользователя, поэтому будет правильно отправлять запрос с параметром `follow=True`.

login, logout и force_login

Вместо заполнения формы авторизации можно применить метод `login`: он авторизует пользователя по логину и паролю:

```
>>> c = Client()
>>> c.login(username='terminator', password='skynetMyLove')
True
```

Если авторизация прошла успешно, то метод вернет `True`.

При тестировании можно авторизовать пользователя даже без пароля, обратившись к объекту пользователя методом `force_login()`

```
>>> c = Client()
>>> user = User.objects.get(username="terminator")
>>> c.force_login(user)
True
```

Для тестирования событий, происходящих после выхода пользователя из системы, можно вызвать метод `logout()`. Но если вам это не нужно для тестирования — разлогинивать пользователей совершенно не обязательно: пользовательская сессия удаляется после завершения тестов.

Информация об обращении

На любой запрос клиента возвращается специальный `response`-объект. В нём содержится ответ сервера и некоторые дополнительные свойства:

- `client` — объект клиента, который использовался для обращения

- `content` — данные ответа в виде строки байтов
- `context` — словарь переменных, переданный для отрисовки шаблона при вызове функции `render()`
- `request` — объект `request`, первый параметр `view`-функций
- `templates` — перечень объектов шаблонов, вызванных для отрисовки страницы, возвращаемой сервером
- `resolver_match` — специальный объект, соответствующий объекту `path()` из списка `urlpatterns`

Пример теста, который использует возможности клиента и ответов сервера:

```
class ProfileTest(TestCase):
    def setUp(self):
        # создание тестового клиента – подходящая задача для функции
setUp()
        self.client = Client()
        # создаём пользователя
        self.user = User.objects.create_user(
            username="sarah", email="connor.s@skynet.com",
password="12345"
        )
        # создаём пост от имени пользователя
        self.post = Post.objects.create(text="You're talking about
things I haven't done yet in the past tense. It's driving me crazy!",
author=self.user)

    def test_profile(self):
        # формируем GET-запрос к странице сайта
        response = self.client.get("/sarah/")

        # проверяем что страница найдена
        self.assertEqual(response.status_code, 200)

        # проверяем, что при отрисовке страницы был получен список из
1 записи
        self.assertEqual(len(response.context["posts"]), 1)

        # проверяем, что объект пользователя, переданный в шаблон,
        # соответствует пользователю, которого мы создали
        self.assertIsInstance(response.context["profile"], User)
        self.assertEqual(response.context["profile"].username,
self.user.username)
```

Особенности работы с базой данных

При тестировании Django создает временную версию базы в памяти, а не обращается к существующей базе. Все записи и изменения, которые вносятся в базу во время тестов, производятся именно с этой временной версией, а основная база проекта остается в исходном состоянии.

При обычной работе в интерактивном режиме вы обращаетесь к базе реального проекта, а во время запуска тестов — ко временной базе.

34_Паджинатор

После запуска наш проект Poemnotes разрастётся, количество записей в лентах увеличится, и если вывести все записи на одной странице — сайтом пользоваться станет невозможно.

Пока что на страницы нашего проекта выводится ограниченное число записей. Даже если на сайте опубликовано несколько тысяч постов — пользователь сможет прочесть только одиннадцать из них.

Код view-функции для главной страницы сейчас выглядит примерно так:

```
def index(request):
    latest = Post.objects.order_by('-pub_date')[:11]
    return render(request, 'index.html', {"posts": latest})
```

Проблема вывода большого числа постов решается разделением ленты на отдельные страницы с ограниченным числом записей на каждой. Для перехода между страницами добавляются специальные ссылки.

Список ссылок для постраничного перехода вам знаком, это стандартный элемент интерфейсов сайтов. Такой компонент есть и во фреймворке Bootstrap:

Такую полосу можно достаточно просто реализовать самому.

- При постраничном делении передать в GET-запросе переменную `page`, значением которой будет номер запрошенной страницы, например `/leo?page=8`
- Проверить, есть ли переменная `page` в GET-параметрах
- Если нет — отдавать первую страницу, с постами с первого по одиннадцатый.
- Если есть
 - Посчитать количество записей в базе
 - С помощью `OFFSET` и `LIMIT` получить нужный диапазон записей и отобразить их на странице

Но писать такой код для каждого списка элементов на сайте было бы неправильно, надо автоматизировать эту задачу.

В Django эту проблему решает стандартный модуль **Paginator**: он «раскладывает» списки по отдельным страницам, выводя на каждую страницу требуемое количество элементов. Для названия этой системы в русском языке используют кальку с английского и называют её «паджинатор».

```
>>> from django.core.paginator import Paginator
# создаём тестовый список
>>> items = ['Антон Чехов', 'Владимир Набоков', 'Лев Толстой', 'Марина
Цветаева']

# создаём объект Paginator(object_list, per_page), зададим деление по два
# объекта на страницу
>>> p = Paginator(items, 2)

# свойство count показывает, сколько объектов в последовательности
>>> p.count
4

# свойство num_pages показывает сколько страниц получится из списка
# num_pages рассчитывается как len(items)/per_page
>>> p.num_pages
2

# получаем объект с элементами для первой страницы
>>> page1 = p.page(1)
>>> page1
<Page 1 of 2>

# получаем элементы для отображения на первой странице
```

```

>>> page1.object_list
['Антон Чехов', 'Владимир Набоков']

# проверяем, есть ли следующие страницы после текущей:
# надо ли отображать кнопку "Следующая страница"
>>> page1.has_next()
True
# проверяем, есть ли страницы перед текущей:
# надо ли отображать кнопку "Предыдущая страница"
>>> page1.has_previous()
False

>>> page2 = p.page(2)
>>> page2.object_list
['Лев Толстой', 'Марина Цветаева']

>>> page2.has_next()
False
>>> page2.has_previous()
True

# чтобы отобразить список с номерами доступных страниц
# получим значение свойства page_range:
# в нём хранятся данные типа range
>>> type(p.page_range)
<class 'range'>
# выведем в консоль линейку с перечнем страниц
>>> for n in p.page_range:
...     print(f"<{n}> ", end="")
...
<1> <2>

```

Объект **Paginator** получает на вход последовательность, разбивает ее на отдельные страницы и позволяет обратиться к ним индивидуально или получить полный список получившихся страниц.

При запросе данных из базы Django ORM тоже возвращает последовательность, и работать с ней можно точно так же, как со списком писателей, который мы только что создали в примере.

```

>>> from posts.models import Post
>>> posts = Paginator(Post.objects.order_by('-pub_date'), 2)
# в переменную post_page передадим объект второй страницы паджинатора
>>> post_page = posts.page(2)
>>> post_page.object_list
<QuerySet [<Post: 36>, <Post: 35>]>

```

Свойства объекта страницы `post_page`:

- `post_page` — значение `<Page 2 of 19>`, тип `django.core.paginator.Page`
- `post_page.has_next()` — значение `True`, тип `bool`

- `post_page.has_previous()` — значение `True`, тип `bool`
- `post_page.has_other_pages()` — значение `True`, тип `bool`
- `post_page.next_page_number()` — значение `3`, тип `int`
- `post_page.previous_page_number()` — значение `1`, тип `int`
- `post_page.start_index()` — номер первого элемента на текущей странице от начала списка, если считать с `1`, значение `3`, тип `int`
- `post_page.end_index()` — номер последнего элемента на текущей странице от начала списка, если считать с `1`, значение `4`, тип `int`

Методы `.start_index()` и `.end_index()` нужны для того, чтобы нумеровать элементы списка

Применение паджинатора

Обновим view-функцию главной страницы:

```
from django.core.paginator import Paginator
from django.shortcuts import render

from .models import Post

def index(request):
    post_list = Post.objects.order_by('-pub_date').all()
    paginator = Paginator(post_list, 10) # показывать по 10 записей на странице.

    page_number = request.GET.get('page') # переменная в URL с номером запрошенной страницы
    page = paginator.get_page(page_number) # получить записи с нужным смещением
    return render(
        request,
        'index.html',
        {'page': page, 'paginator': paginator}
    )
```

Также надо обновить шаблон главной страницы чтобы работать не со списком `posts`, который использовался раньше, а с новым объектом `page`:

```
{% extends "base.html" %}
{% block title %} Последние обновления {% endblock %}
{% block content %}

<h1> Последние обновления на сайте<h1>

{% for post in page %}
    ...Тут вывод списка записей...
{% endfor %}
```

```

{% if page.has_other_pages %}
    ...Тут код со списком страниц для листания...
{% endif %}

{% endblock %}

```

Остаётся вывести список для перехода по страницам. Таких страниц у нас множество, и чтобы много раз не повторять один и тот же код — напишем виджет, который будем включать в те шаблоны, где он необходим.

В директории `templates` создайте файл `paginator.html` и добавьте в него такой код:

```

<nav aria-label="Переключение страниц">
    <ul class="pagination">
        {% if items.has_previous %}
            <li class="page-item"><a class="page-link" href="?page={{ items.previous_page_number }}">&laquo; Предыдущая</a></li>
        {% else %}
            <li class="page-item disabled"><a class="page-link" href="#" tabindex="-1" aria-disabled="true">&laquo; Предыдущая</a></li>
        {% endif %}
        {% for i in paginator.page_range %}
            {% if items.number == i %}
                <li class="page-item active"><span class="page-link">{{ i }}</span></li>
            {% else %}
                <li class="page-item"><a class="page-link" href="?page={{ i }}">{{ i }}</a></li>
            {% endif %}
        {% endfor %}
        {% if items.has_next %}
            <li class="page-item"><a class="page-link" href="?page={{ items.next_page_number }}">Следующая &raquo;</a></li>
        {% else %}
            <li class="page-item disabled"><a class="page-link" href="#" tabindex="-1" aria-disabled="true">Следующая &raquo;</a></li>
        {% endif %}
    </ul>
</nav>

```

И обновите код файла `index.html`:

```

{% extends "base.html" %}
{% block title %} Последние обновления {% endblock %}
{% block content %}

<h1> Последние обновления на сайте<h1>
{% for post in page %}
    ...Тут вывод списка записей...
{% endfor %}

```

```
{% if page.has_other_pages %}
    {% include "paginator.html" with items=page paginator=paginator %}
{% endif %}
****
```

{% endblock %}

Теперь в любой шаблон, где может потребоваться постраничное деление контента, можно добавить фрагмент кода с переключателем страниц.

35_Вспомогательные страницы flatpages

Вы уже написали несколько страниц и view-функций для сайта. Помимо таких страниц на сайте могут понадобиться простые статичные страницы с текстом:

- Контактная информация
- Правила и условия использования
- Юридическая информация
- Условия размещения рекламы
- ...и множество других

Создавать для каждой такой страницы отдельную view-функцию нерационально: клиент захочет редактировать эти страницы, а привлекать программистов ради исправления пары слов — долго и дорого.

Для управления такими страницами в Django есть приложение **flatpages**. Оно поставляется вместе с Django, вам надо его подключить и настроить.

Добавление приложения

Для подключения приложения добавьте его имя в список INSTALLED_APPS в конфиге сайта. Приложение **flatpages** зависит от приложения **sites**, оно есть в стандартной поставке Django, и его тоже нужно подключить:

```
INSTALLED_APPS = [
    'users',
    'posts',
    'django.contrib.sites',
    'django.contrib.flatpages',
    ...
]
```

Приложение **sites** подключает режим мульти сайтовости и позволяет создавать «подсайты». Это полезно, если нужно создать и вести из одной админки несколько похожих сайтов. Мульти сайтовость нам не понадобится, так что укажем в настройках ID текущего сайта и забудем об этом. Добавьте в файл `poemnotes/settings.py` новую строку:

```
# Идентификатор текущего сайта
SITE_ID = 1
```

В приложениях, которые вы подключили, есть свои модели, для них нужны таблицы в базе данных. Выполните миграции, чтобы всё заработало.

Настройка адресов и шаблонов flatpages

Пусть адрес вспомогательных страницы сайта будет начинаться со строки `/about/`. Добавьте в главный `urls.py` дополнительное правило:

```
urlpatterns = [
    # раздел администратора
    path('admin/', admin.site.urls),
    # flatpages
    path('about/', include('django.contrib.flatpages.urls')),
    # регистрация и авторизация
    path('auth/', include('users.urls')),
    path('auth/', include('django.contrib.auth.urls')),
    # импорт из приложения posts
    path('', include('posts.urls')),
]
```

В директорию `templates` добавьте шаблон для отображения статичных страниц, он должен быть доступен по адресу `templates/flatpages/default.html`:

```
{% extends "base.html" %}
{% block title %} {{ flatpage.title }} {% endblock %}
{% block content %}

<div class="container">
    <h1>{{ flatpage.title }}</h1>
    {{ flatpage.content }}
</div>

{% endblock %}
```

Дополнительные настройки

После добавления приложений **sites** и **flatpages** в интерфейсе администратора добавилось два новых раздела:

Администрирование Django

Администрирование сайта

POSTS

Posts + Добавить Изменить

ПОЛЬЗОВАТЕЛИ И ГРУППЫ

Группы + Добавить Изменить

Пользователи + Добавить Изменить

ПРОСТЫЕ СТРАНИЦЫ

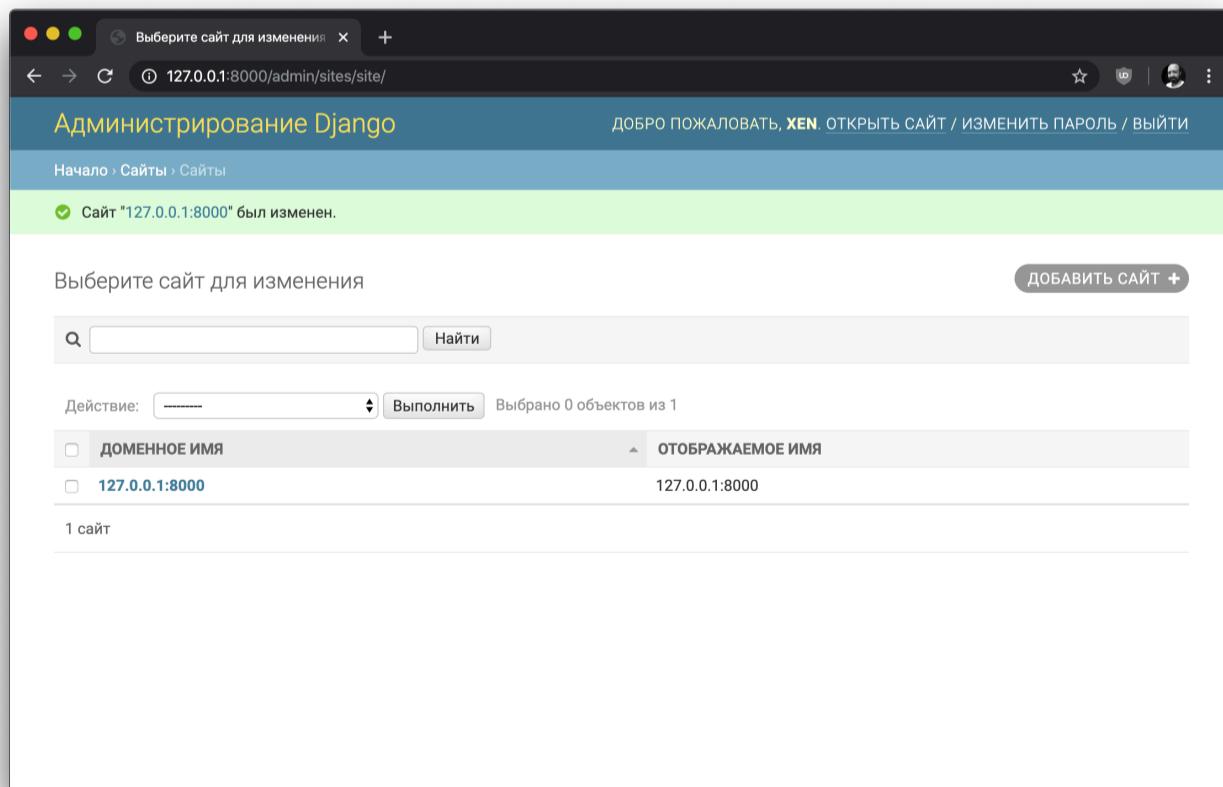
Простые страницы + Добавить Изменить

САЙТЫ

Сайты + Добавить Изменить

Обратите внимание, что SITE_ID, который вы указывали в настройках, должен совпадать с id записи в модели Site. Первая запись там уже добавлена.

Для удобства в поле «Доменное имя» укажите локальный адрес компьютера:



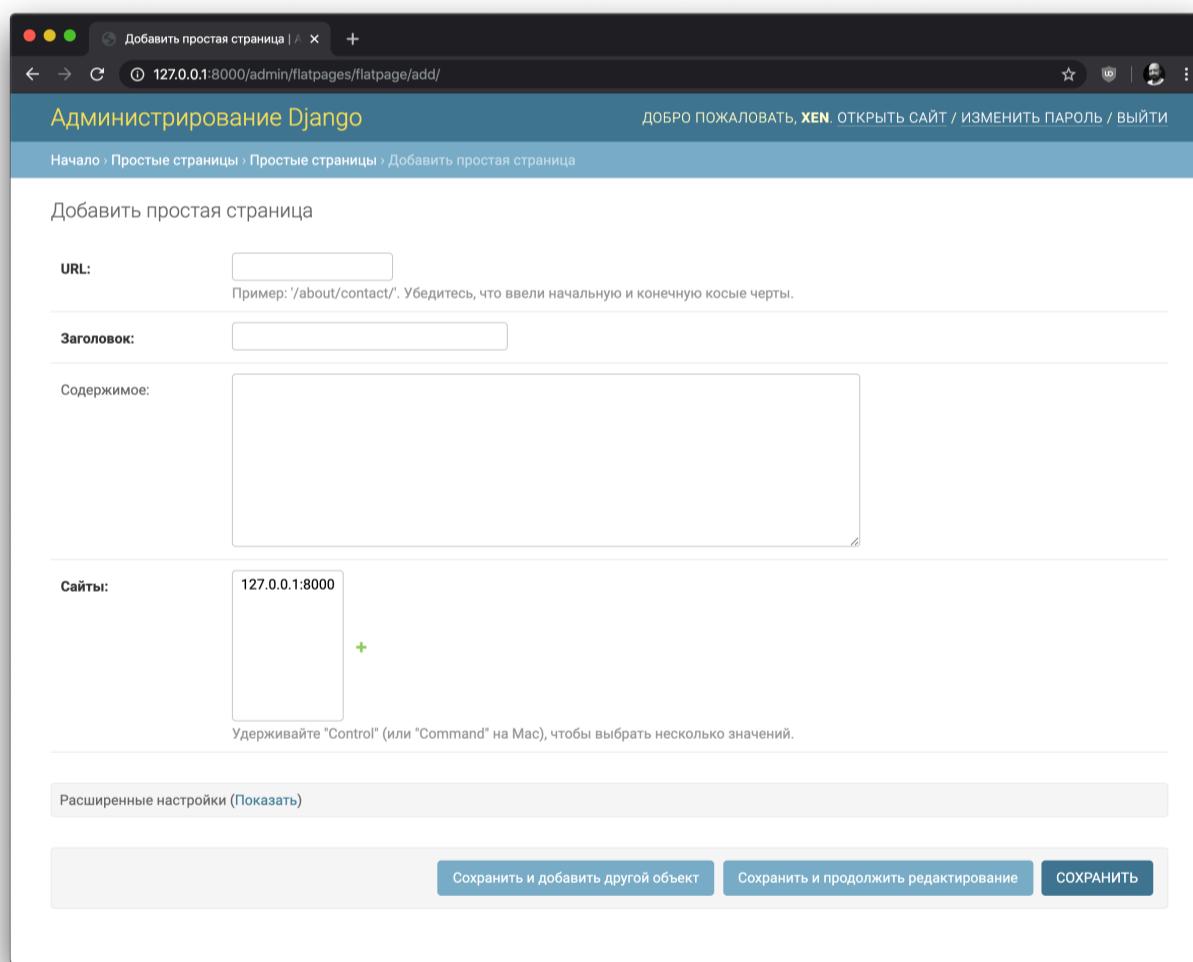
В `urlpatterns` головного `urls.py` мы указали префикс `/about/`. Теперь адреса с этим префиксом маршрутизатор будет отдавать приложению `flatpages`. В `urls.py` приложения `flatpages` видно, что маска пути выглядит следующим образом:

```
urlpatterns = [
    path('<path:url>', views.flatpage,
name='django.contrib.flatpages.views.flatpage'),
]
```

При обращении к статичной странице Django проверит на совпадение путь `/about/`, и оставшуюся часть адреса передаст во view-функцию `flatpage()` в переменной `url`.

Создание первой flatpage

В интерфейсе администратора перейдите к форме создания новой страницы:



В поле **URL** указывается уникальный в пределах сайта адрес без префикса `/about`

Создаваемая страница через поле **Сайты** привязывается к определённому сайту и показывается только на нём. Это важно для мультисайтowego проекта, а нам проще: в системе только один сайт.

Создайте страницу обратной связи и сохраните её:

The screenshot shows the Django admin interface for creating a new flatpage. The URL is `127.0.0.1:8000/admin/flatpages/flatpage/1/change/`. The page title is "Изменить простая страница". The URL field contains `/contacts/`. The title field is "Контактная информация". The content field contains the following HTML code:

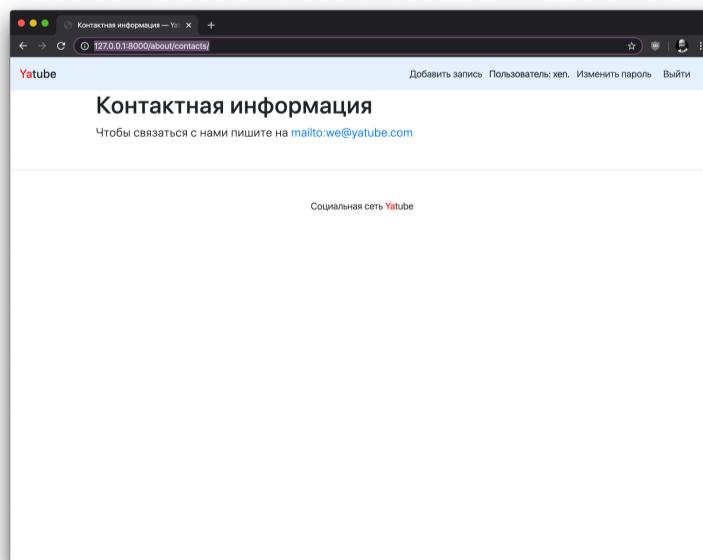
```
<p class="lead">Чтобы связаться с нами пишите на <a href="mailto:we@yatube.com">mailto:we@yatube.com</a></p>
```

The "Sites" field is set to `127.0.0.1:8000`. At the bottom, there are buttons: "Удалить" (Delete), "Сохранить и добавить другой объект" (Save and add another object), "Сохранить и продолжить редактирование" (Save and continue editing), and a large blue "СОХРАНИТЬ" (Save) button.

Поле **содержимое** позволяет вставлять HTML-код. На практике это приводит к тому, что неопытные администраторы сайта умудряются вставить код с незакрытыми или перепутанными тегами и полностью сломать отображение страницы. Такая ситуация встречается гораздо чаще, чем можно предположить.

При попытке кликнуть на ссылку "Смотреть на сайте" в новом окне откроется страница `http://127.0.0.1:8000/contacts/` (без префикса `/about`) и появится сообщение об ошибке 404. Дело в том, что приложение `flatpages` «не знает», что его страницы расположены в разделе `/about`.

Удостоверьтесь, что страница открывается по адресу `http://127.0.0.1:8000/about/contacts/`



Приложение **flatpages** — не лучшая из систем для создания статичных страниц, но это простой предустановленный инструмент, дающий представление о таких приложениях.

Использование flatpages для предварительно заданных страниц

Страницы **flatpages** можно создавать с любыми адресами, выходя за пределы предустановленной структуры `/about/page-name`. Допустим, на сайте должны быть страницы «О нас» `/about-us` и «Условия использования» `/terms`, но их адреса не должны предваряться префиксом `/about` (для того, чтобы жизнь была веселее, с вами будут работать шаманы из SEO, специалисты по продвижению; зачастую именно они ставят условия по структуре и именам в URL).

Добавим в список `urlpatterns` головного `urls.py` адреса этих страниц:

```
from django.contrib.flatpages import views
urlpatterns = [
    # тут все path()
]

# добавим новые пути
urlpatterns += [
    path('about-us/', views.flatpage, {'url': '/about-us/'},
name='about'),
    path('terms/', views.flatpage, {'url': '/terms/'}, name='terms'),
]
```

Теперь страницы с адресами `/about-us` и `/terms` будут обрабатываться view-функцией `flatpage()` приложения **flatpages**.

`{'url': '/about-us/'}` и `{'url': '/terms/'}` — это параметры, которые `path()` передаёт в вызываемую view-функцию. Это даёт нам свободу: например, мы можем обработать URL `/rasskaz-o-tom-kakie-my-horoshie` точно так же, как и URL `/about-us`, «подменив» передаваемый во view-функцию параметр `url`

```
urlpatterns = [
    path('rasskaz-o-tom-kakie-my-horoshie/', views.flatpage, {'url': '/about-us/'}, name='about'),
]
```

Когда обработка «эксклюзивных» URL подготовлена — можно в админ-зоне создать две страницы, указав для них URL `/about-us/` и `/terms/` соответственно, и работать с ними, как с обычными *flatpages*.

Тестирование flatpages

Для полноценного покрытия сайта тестами вам понадобится создавать страницы программно. Для этого можно импортировать модель из модуля `django/contrib/flatpages/models.py` и перед тестированием создавать объекты. Точно так же можно работать и с моделью `Site` из `django/contrib/sites/models.py`.

Изменение сайта

Измените файл `templates/footer.html` следующим образом:

```
<footer class="pt-4 my-md-5 pt-md-5 border-top">
    <p class="m-0 text-dark text-center "><a href="/about-author/">06
авторе</a> - <a href="/about-spec/">Технологии</a></p>
    <p class="m-0 text-dark text-center ">Социальная сеть <span
style="color:red">Poem</span>notes</p>
</footer>
```

Добавьте в `urls.py` поддержку адресов страниц «Об авторе» `/about-author` и «Технологии» `/about-spec`.

Через админ-зону создайте страницу «Об авторе» и опубликуйте на ней информацию о себе со ссылками на ваш Github-профайл и на страницу вашего резюме, расскажите, что вас вдохновляет в текущий момент.

Вторую страницу назовите «Технологии»: расскажите на ней, какие программные инструменты вы применили для создания этого сайта.

Если во время работы вам помогал ваш кот, собака или другое домашнее животное — упомяните об этом: в среде программистов принято это указывать, без этого вас просто не возьмут на работу в приличную компанию! Настоящие хакеры еще выкладывают фотографию своей рабочей кружки (картинку можно положить в папку `static`, но не забудьте уменьшить размер файла до приемлемого).

37_Профайл пользователя и страница записи

Создадим страницу профайла пользователя. Пока что на ней будет отображаться информация об авторе и его посты.

Дополнительно напишем страницу для просмотра отдельного поста, а в следующих уроках добавим на неё форму для комментариев и систему подписки на автора.

Персональная страница

В качестве адреса персональной страницы автора будет использоваться его *username*, это логично и удобно. Добавьте в конец файла `posts/urls.py` новые пути:

```
from django.urls import path
from . import views

urlpatterns = [
    ...
    # Главная страница
    path('', views.index, name='index'),
    # Профайл пользователя
    path('<str:username>', views.profile, name='profile'),
    # Просмотр записи
    path('<str:username>/<int:post_id>', views.post_view, name='post'),
    path(
        '<str:username>/<int:post_id>/edit/',
        views.post_edit,
        name='post_edit'
    ),
]
```

А в файл `posts/views.py` добавьте функции:

```
def profile(request, username):
    # тут тело функции
    return render(request, 'profile.html', {})

def post_view(request, username, post_id):
    # тут тело функции
    return render(request, 'post.html', {})

def post_edit(request, username, post_id):
    # тут тело функции. Не забудьте проверить,
    # что текущий пользователь – это автор записи.
    # В качестве шаблона страницы редактирования укажите шаблон создания
    # новой записи
    # который вы создали раньше (вы могли назвать шаблон иначе)
    return render(request, 'post_new.html', {})
```

Шаблон страницы профайла

Обычно backend-разработчик берет готовый HTML-код либо комбинирует части существующих шаблонов. Сейчас ваша задача — превратить статичный HTML-код в динамический шаблон.

В этом фрагменте HTML-кода выводятся посты определённого пользователя и информация о нём. Этот блок **content** должен встраиваться в базовый шаблон **base.html** (подсказку про тег `{% extends "file_name.html" %}` мы не дадим).

Комментарии в коде показывают, где статические данные нужно заменить на переменные.

```
<main role="main" class="container">
    <div class="row">
        <div class="col-md-3 mb-3 mt-1">
            <div class="card">
                <div class="card-body">
                    <div class="h2">
                        <!-- Имя автора -->
                        Лев Толстой
                    </div>
                    <div class="h3 text-muted">
                        <!-- username автора -->
                        @leo
                    </div>
                </div>
                <ul class="list-group list-group-flush">
                    <li class="list-group-item">
                        <div class="h6 text-muted">
                            Подписчиков: XXX <br />
                            Подписан: XXX
                        </div>
                    </li>
                    <li class="list-group-item">
                        <div class="h6 text-muted">
                            <!-- Количество записей -->
                            Записей: 36
                        </div>
                    </li>
                </ul>
            </div>
        <div class="col-md-9">
            <!-- Начало блока с отдельным постом -->
            <div class="card mb-3 mt-1 shadow-sm">
                <div class="card-body">
                    <p class="card-text">
```

```

        <!-- Ссылка на страницу автора в
атрибуте href; username автора в тексте ссылки -->
        <a href="/leo/"><strong class="d-
block text-gray-dark">@leo</strong></a>
        <!-- Текст поста -->
        [Фокшаны.] Еще переходъ до
Фокшанъ, во время котораго я ъхаль съ Монго. Человѣкъ пустой, но съ твердыми,
хотя и ложными убѣжденіями. Генерал[у] по этому должно быть случаю, угодно
было спрашивать о моемъ здоровьи. Свинья! К[о]выряль носъ и ничего не написаль
– вотъ 2 упрека за нын[ъшній] день. Послѣдній упрекъ становится слишкомъ
часть, хотя походъ и можетъ служить въ немъ отчасти извиненіемъ. Отношенія мои
съ товарищами становятся такъ пріятны, что мнѣ жалко бросить штабъ. Здоровье
кажется (2) лучше.

        </p>
        <div class="d-flex justify-content-between
align-items-center">
            <div class="btn-group ">
                <!-- Ссылка на страницу
записи в атрибуте href-->
                <a class="btn btn-sm text-
muted" href="/leo/37/" role="button">Добавить комментарий</a>
                <!-- Ссылка на
редактирование, показывается только автору записи -->
                <a class="btn btn-sm text-
muted" href="/leo/37/edit" role="button">Редактировать</a>
            </div>
            <!-- Дата публикации -->
            <small class="text-muted">31 июля
1854 г. 0:00</small>
        </div>
    </div>
    <!-- Конец блока с отдельным постом -->

    <!-- Остальные посты -->

    <!-- Здесь постраничная навигация паджинатора -->

```

Шаблон страницы просмотра записи

Этот шаблон почти идентичен странице профайла, но в нём показывается только один пост.

```

<main role="main" class="container">
    <div class="row">
        <div class="col-md-3 mb-3 mt-1">
            <div class="card">
                <div class="card-body">
                    <div class="h2">

```

```
<!-- Имя автора -->
Лев Толстой
</div>
<div class="h3 text-muted">
    <!-- username автора -->
    @leo
</div>
</div>
<ul class="list-group list-group-flush">
    <li class="list-group-item">
        <div class="h6 text-muted">
            Подписчиков: XXX <br />
            Подписан: XXX
        </div>
    </li>
    <li class="list-group-item">
        <div class="h6 text-muted">
            <!--Количество записей -->
            Записей: 36
        </div>
    </li>
</ul>
</div>
</div>

<div class="col-md-9">
    <!-- Пост -->
    <div class="card mb-3 mt-1 shadow-sm">
        <div class="card-body">
            <p class="card-text">
                <!-- Ссылка на страницу автора в
атрибуте href; username автора в тексте ссылки -->
                <a href="/leo/"><strong class="d-block
text-gray-dark">@leo</strong></a>
                <!-- Текст поста -->
                [Фокшаны.] Еще переходъ до Фокшанъ, во
время котораго я ъхалъ съ Монго. Человѣкъ пустой, но съ твердыми, хотя и
```

ложными убѣжденіями. Генерал[у] по этому должно быть слушаю, угодно было спрашивать о моемъ здоровыи. Свинья! К[о]вырять носъ и ничего не написаль – вотъ 2 упрека за нын[ѣшній] день. Послѣдній упрекъ становится слишкомъ часть, хотя походъ и можетъ служить въ немъ отчасти извиненiemъ. Отношенія мои съ товарищами становятся такъ пріятны, что мнѣ жалко бросить штабъ. Здоровье кажется (2) лучше.

```
</p>
<div class="d-flex justify-content-between align-items-center">
    <div class="btn-group">
        <!-- Ссылка на редактирование,
показывается только автору записи -->
        <a class="btn btn-sm text-muted" href="/leo/36/edit" role="button">Редактировать</a>
    </div>
    <!-- Дата публикации -->
    <small class="text-muted">31 июля 1854
г. 0:00</small>
</div>
</div>
</div>
</main>
```

Задание

Создайте в своём локальном проекте **Poemnotes**:

- Страницу профайла пользователя с постами. Добавьте на неё паджинатор, количество записей автора вы умеете вычислять
- Страницу просмотра отдельной записи
- Страницу с формой редактирования существующей записи: расширьте готовый шаблон с формой создания поста. В зависимости от ситуации заголовок формы должен меняться: «Добавить запись» или «Редактировать запись». Надпись на кнопке отправки формы тоже должна зависеть от операции: «Добавить» для новой записи и «Сохранить» — для редактирования.

38_Страницы с ошибками

Режим отладки проекта

Сейчас ваш проект работает **в режиме отладки**: при установке проекта в конфиге был автоматически выставлен флаг `DEBUG=True`. В этом режиме при ошибках выводится страница с технической информацией и подробным разбором строк, в которых что-то не так.

Пользователям такую страницу показывать нельзя. Во-первых, это некрасиво, непонятно и бессмысленно: пользователю не нужна эта информация. Во-вторых, это небезопасно: в отладочной информации могут содержаться ключи доступа к внешним сервисам или к базе данных. В-третьих, исследование причин ошибок не входит в задачи посетителей.

Так что не забудьте отключить режим отладки при публикации сайта на боевом сервере.

Страницы ошибок

Если отключить режим отладки (его ещё называют «режим разработки» или «режим разработчика»), то часть страниц вы увидите в совершенно ином виде.

Страницы ошибок (`error 404`, «страница не найдена» или `error 500`, «ошибка сервера») предустановлены в Django, но выглядят так, как будто разработчику сайта не было до них дела.

Давайте это исправим.

Если запрошенная страница не найдена, сервер возвращает код 404. Django видит этот ответ и автоматически вызывает собственную предустановленную view-функцию. Адрес этой view-функции хранится в переменной `handler404` (по умолчанию это view-функция `django.views.defaults.page_not_found`).

Но можно создать view-функцию самостоятельно и вызвать её при ошибке 404. Для этого надо лишь перезаписать содержимое переменной `handler404`: передать в эту переменную имя нашей view-функции.

Точно так же дело обстоит и с обработкой прочих ошибок: для них заготовлены переменные `handler400`, `handler403` и несколько других, в документации есть их перечень.

Импортируем переменные из модуля `django.conf.urls`, переопределим дефолтные переменные — и сможем вызвать собственные view-функции и шаблоны для страниц ошибок.

Указывать view-функции для таких страниц можно только в головном urls.py. Добавьте в него следующие строки:

```
from django.conf.urls import handler404, handler500

handler404 = "posts.views.page_not_found"
handler500 = "posts.views.server_error"
```

Если в вашем редакторе кода включён анализ синтаксиса, возможно, вы получите предупреждение о том, что переменные созданы, но не используются. Но мы-то знаем, что это не так.

Чтобы редактор не приставал к вам по пустякам, выполните трюк, который отключит проверку текущей строки — добавьте в неё директиву-комментарий `# noqa` (от *NO Quality Assurance*):

```
from django.conf.urls import handler404, handler500

handler404 = "posts.views.page_not_found" # noqa
handler500 = "posts.views.server_error" # noqa
```

View-функции можно положить в любое удобное место, мы сохраним их в файле posts/views.py:

```
def page_not_found(request, exception):
    # Переменная exception содержит отладочную информацию,
    # выводить её в шаблон пользовательской страницы 404 мы не станем
    return render(
        request,
        "misc/404.html",
        {"path": request.path},
        status=404
    )

def server_error(request):
    return render(request, "misc/500.html", status=500)
```

Создайте файл шаблона templates/misc/404.html и добавьте в него код:

```
{% extends "base.html" %}
{% block title %} Ошибка 404 {% endblock %}
{% block content %}

<main role="main" class="container">
<div class="row">
    <div class="col-md-12">
        <h1>Ошибка 404</h1>
        <p class="lead">Страница <code>{{ path }}</code> не найдена</p>
        <p class="lead"><a href="{% url "index" %}">Вернуться на главную</a></p>
```

```
</div>
</div>
</main>

{% endblock %}
```

Теперь надо создать шаблон для ошибки 500, `templates/misc/500.html`

```
{% extends "base.html" %}

{% block title %} Ошибка 500 {% endblock %}

{% block content %}

<main role="main" class="container">
<div class="row">
    <div class="col-md-12">
        <h1>Ошибка 500</h1>
        <p class="lead">Ошибка на сервере, попробуйте обновить страницу или
обратиться позже</p>
        <p class="lead"><a href="{% url "index" %}">Вернуться на главную</a></
p>
    </div>
</div>
</main>

{% endblock %}
```

Директорию `templates/misc` создавать не обязательно, но с ней будет удобнее: там можно спрятать файлы, которые практически никогда не изменяются. Так они не будут мешать при работе с другими шаблонами.

Включение и отключение режима отладки

При разработке реальных проектов вы будете публиковать их на сервере, и режим отладки нужно будет отключать. Чтобы это сделать — измените в файле `settings.py` значение ключа `DEBUG` на **False**.

Если обратиться к страницам сайта, размещённого на удалённом сервере, то, вполне вероятно, вы получите сообщение об ошибке: код *400 Bad Request*. Это может поставить в тупик, если не учесть, что причиной ошибки бывает отсутствие в списке `ALLOWED_HOSTS` адреса вашего боевого сервера.

После выполнения заданий из предыдущих уроков ALLOWED_HOSTS у вас должен быть в порядке, но проверьте его ещё раз:

```
DEBUG = False

ALLOWED_HOSTS = [
    "localhost",
    "127.0.0.1",
    "[::1]",
    "testserver",
]
```

После отключения режима разработки у вас пропадут некоторые дополнительные функции, зато вы сможете проверить работу страниц ошибок.

Для этого просто добавьте их в urls.py, например, с адресами /404 и /500.

После проверки работы шаблонов не забудьте вернуть сайт в режим разработки:

```
DEBUG = True
```

39_Добавление картинок к постам

Django — это огромная экосистема из всевозможных модулей, расширяющих возможности базового фреймворка. На сайте PyPI.org размещены десятки тысяч расширений, в названии которых есть слово django, а неопубликованных модулей или тех, что названы как-то иначе — ещё больше.

Пустим в дело богатства экосистемы Django: подключим к нашему проекту управление изображениями.

sorl-thumbnail

В стандартную установку Django встроен инструмент для работы с картинками, но задачи вроде изменения размера изображений ему не по силам. Django умеет только загружать файлы и отдавать их как есть.

Для настоящей соцсети этого мало. Пользователи могут залить RAW-картинку с фотоаппарата размером в 5950×3968 пикселей и весом в 50 Мб, или выложить картинку-мем размером в 120x80. Если опубликовать эти

картинки как есть — сайт будет выглядеть неопрятно или долго загружаться.

Дадим пользователям возможность иллюстрировать посты и сделаем так, чтобы загруженные изображения выглядели более-менее одинаково.

Одно из первых по популярности и удобству приложений для работы с графикой — [sorl-thumbnail](#). Для его работы нужна графическая библиотека, **sorl-thumbnail** умеет работать со многими, мы возьмём библиотеку **Pillow**.

Когда-то, когда Python еще набирал популярность, в компании Secret Labs AB написали библиотеку PIL (от Python Imaging Library). Это была одна из первых графических библиотек, предназначенных именно для Python. Она быстро стала стандартом в сообществе. Некоторое время спустя компания перестала существовать и развивать библиотеку: последняя версия PIL ориентирована на Python v.2.3. Но потребность в обработке изображений никуда не пропала, и Alex Clark сделал форк (ответвление проекта) под новым названием Pillow (англ. «подушка»). Очевидно, что автор дал библиотеке название, включающее буквы pil, в честь старого проекта, а не потому, что любит спать. Новая библиотека поддерживает совместимость и для старых проектов.

Для установки Pillow выполните команду в виртуальном окружении проекта:

```
(venv) $ pip install Pillow
```

Возможно, вам понадобится [установить дополнительные библиотеки для вашей операционной системы](#).

Теперь установите приложение `sorl-thumbnail`:

```
(venv) $ pip install sorl-thumbnail
```

Добавьте приложение в список `INSTALLED_APPS`, в конец списка:

```
INSTALLED_APPS = [  
    # ...  
    'sorl.thumbnail',
```

]

Выполните миграцию, и после этого приложение будет готово к работе.
Теперь вам станут доступны специальные теги в шаблонах:

```
<!-- Загрузка тегов библиотеки в шаблон -->
{% load thumbnail %}

<!-- Пример использования тега для пропорционального уменьшения и обрезки
картинки до размера 100x100px с центрированием -->
<!-- и вставляет в код изображение которое отобразится пользователю. -->
{% thumbnail item.image "100x100" crop="center" as im %}
    
    {% load thumbnail %}
    {% thumbnail post.image "960x339" crop="center" upscale=True as im %}
          
...
```

Если в посте нет картинки, то содержимое тега `thumbnail` будет проигнорировано, так что проверку `{% if post.image %}...{% endif %}` делать не надо.

В HTML-форме создания и редактирования поста появится поле для загрузки изображения. Форма должна понимать, что из неё на сервер будут передаваться файлы. Обновите шаблон с формой — в тег `<form>` добавьте атрибут `enctype`:

```
<form method="post" enctype="multipart/form-data">
```

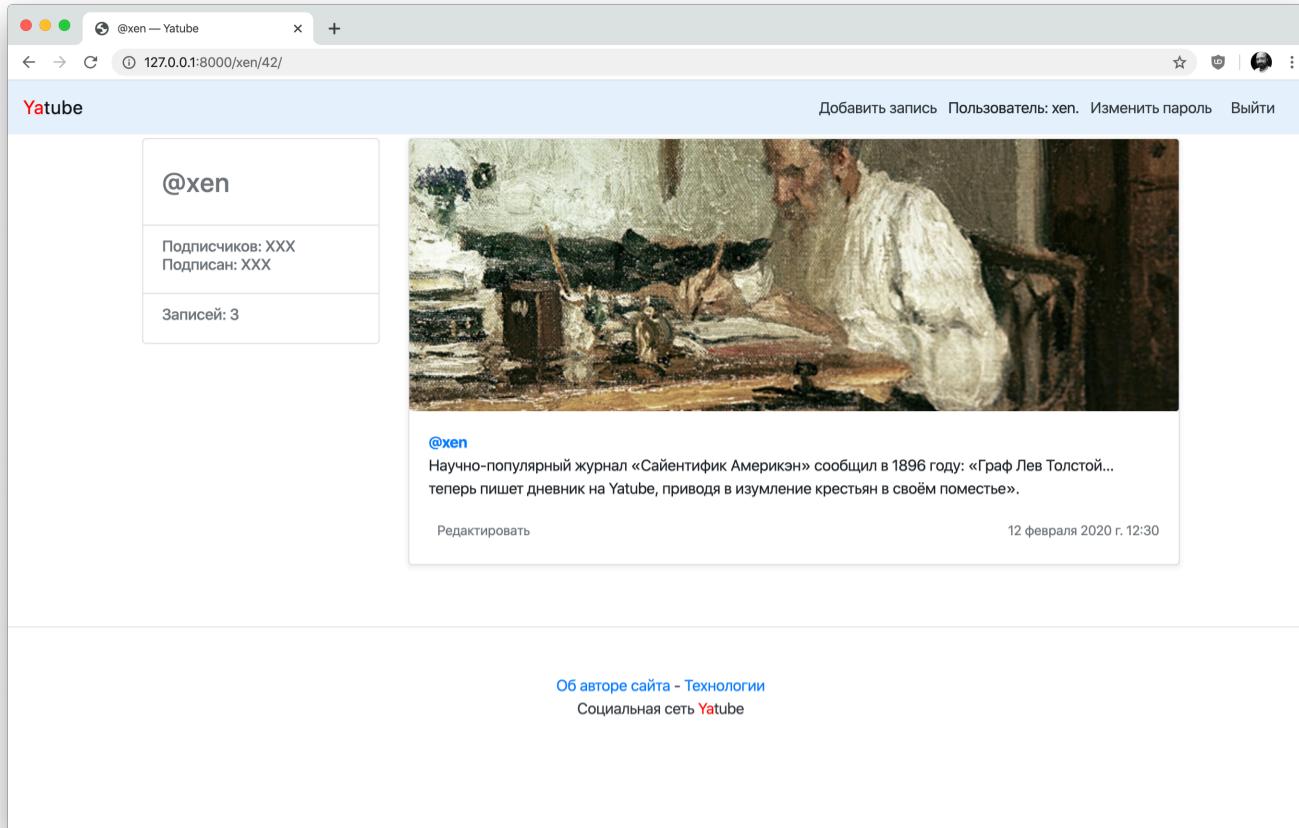
Обновление view-функции

Осталось подправить функцию редактирования записи. Django-формы умеют работать с файлами, так что нужно лишь передать дополнительный параметр `files=request.FILES or None`, и больше ничего! Вам не надо отдельно сохранять файлы, не надо проверять их тип или беспокоиться, что в директории загрузки окажется файл с таким же именем (Django сам переименует файл при необходимости):

```
@login_required  
def post_edit(request, username, post_id):  
    profile = get_object_or_404(User, username=username)  
    post = get_object_or_404(Post, pk=post_id, author=profile)  
    if request.user != profile:  
        return redirect('post', username=username, post_id=post_id)  
    # добавим в form свойство files  
    form = PostForm(request.POST or None, files=request.FILES or None,  
instance=post)  
  
    if request.method == 'POST':  
        if form.is_valid():  
            form.save()  
            return redirect("post", username=request.user.username,  
post_id=post_id)  
  
    return render(  
        request, 'post_edit.html', {'form': form, 'post': post},  
    )
```

Результат

У вас должна получиться страница просмотра записи с картинкой:



40_ Комментарии

В базе данных проекта **Poemnotes** уже хранится информация об авторах и их постах. Дадим пользователям возможность комментировать записи друг друга.

Задание

Напишите систему комментирования записей. На странице просмотра записи под текстом поста выведите форму для отправки комментария, а ниже — список комментариев.

Модель

Добавьте модель **Comment** с такими полями:

- *post* — ссылка на пост, к которому оставлен комментарий (для связи модели **Post** с комментариями используйте имя `comments`).
- *author* — ссылка на автора комментария (для связи модели **User** с комментариями используйте имя `comments`).
- *text* — текст комментария.

- *created* — автоматически подставляемые дата и время публикации комментария.

В случае, если автор комментария или пост будут удалены — все привязанные к ним комментарии должны автоматически удаляться.

Шаблон

Создайте шаблон `comments.html` и подключайте его на странице просмотра записи:

```
<!-- Форма добавления комментария -->
{% load user_filters %}

{% if user.is_authenticated %}
<div class="card my-4">
<form
    action="{% url 'add_comment' post.author.username post.id %}"
    method="post">
    {% csrf_token %}
    <h5 class="card-header">Добавить комментарий:</h5>
    <div class="card-body">
        <form>
            <div class="form-group">
                {{ form.text|addclass:"form-control" }}
            </div>
            <button type="submit" class="btn btn-primary">Отправить</button>
        </form>
    </div>
</form>
</div>
{% endif %}

<!-- Комментарии -->
{% for item in items %}
<div class="media mb-4">
<div class="media-body">
    <h5 class="mt-0">
        <a
            href="{% url 'profile' item.author.username %}"
            name="comment_{{ item.id }}"
            >{{ item.author.username }}</a>
    </h5>
    {{ item.text }}
</div>
</div>

{% endfor %}
```

Роутинг и view

Создайте `path()` и `view`-функцию для обработки отправленного комментария:

```
path("<username>/<int:post_id>/comment", views.add_comment,  
name="add_comment"),
```

Подсказка

Вам надо создать модель `Comment`, `view`-функцию для обработки отправленных комментариев и форму. Для текста комментария в форме должен быть HTML-элемент `<textarea>`.

Измените `view`-функцию страницы просмотра поста: добавьте в неё вывод комментариев.

41_Рефакторинг шаблонов

У нас накопилась достаточно большая библиотека шаблонов, в которых выводятся записи:

- Главная страница
- Страница сообщества
- Страница профайла
- Страница просмотра отдельной записи

В шаблонах повторяются похожие куски кода, которые выводят текст записи с картинкой и ссылками на дополнительные элементы. Для создания новой страницы вы копировали один и тот же фрагмент шаблона. Но если захочется отредактировать HTML-код, в котором выводится пост — вам придется внести исправления в несколько разных шаблонов и наверняка вы забудете о каком-нибудь из них.

Если у разработчика уходит много времени на повторяющиеся действия — это неправильно. Состояние проекта, когда уже отчетливо чувствуется, что пришло время навести порядок в коде, называется «технический долг». Это естественный этап развития проекта, не зря для него придумали отдельный термин.

Проведём рефакторинг проекта. Для начала вынесем код отдельного поста в собственный шаблон, и затем будем встраивать его всюду, где нужно вывести пост.

Создайте шаблон post_item.html с таким содержимым:

```
<div class="card mb-3 mt-1 shadow-sm">

    <!-- Отображение картинки --&gt;
    {%- load thumbnail %}
    {%- thumbnail post.image "960x339" crop="center" upscale=True as im %}
    &lt;img class="card-img" src="{{ im.url }}" /&gt;
    {%- endthumbnail %}

    <!-- Отображение текста поста --&gt;
    &lt;div class="card-body"&gt;
        &lt;p class="card-text"&gt;
            <!-- Ссылка на автора через @ --&gt;
            &lt;a name="post_{{ post.id }}" href="{% url 'profile' post.author.username %}"&gt;
                &lt;strong class="d-block text-gray-dark"&gt;@{{ post.author }}&lt;/strong&gt;
            &lt;/a&gt;
            {{ post.text|linebreaksbr }}
        &lt;/p&gt;

        <!-- Если пост относится к какому-нибудь сообществу, то отобразим ссылку на него через # --&gt;
        {% if post.group %}
            &lt;a class="card-link muted" href="{% url 'group' post.group.slug %}"&gt;
                &lt;strong class="d-block text-gray-dark"&gt;#{ post.group.title }&lt;/strong&gt;
            &lt;/a&gt;
        {% endif %}

        <!-- Отображение ссылки на комментарии --&gt;
        &lt;div class="d-flex justify-content-between align-items-center"&gt;
            &lt;div class="btn-group"&gt;
                &lt;a class="btn btn-sm text-muted" href="{% url 'post' post.author.username post.id %}" role="button"&gt;
                    {% if post.comments.exists %}
                        {{ post.comments.count }} комментариев
                    {% else%}
                        Добавить комментарий
                    {% endif %}
                &lt;/a&gt;
            &lt;/div&gt;
        </pre>
```

```
    <!-- Дата публикации поста -->
    <small class="text-muted">{{ post.pub_date }}</small>
</div>
</div>
</div>
```

Теперь во всех шаблонах код отдельной записи замените на `include` нового шаблона:

```
{% include "post_item.html" with post=post %}
```

Переменная **post** — это объект с записью.

Чтобы у вас не слетела верстка — вот шаблон главной страницы проекта со внесёнными изменениями:

```
{% extends "base.html" %}
{% block title %} Последние обновления {% endblock %}

{% block content %}
    <div class="container">
        <h1> Последние обновления на сайте</h1>
        <!-- Вывод ленты записей -->
        {% for post in page %}
            <!-- Вот он, новый include! -->
            {% include "post_item.html" with post=post %}
        {% endfor %}
    </div>

    <!-- Вывод паджинатора -->
    {% if page.has_other_pages %}
        {% include "paginator.html" with items=page paginator=paginator %}
    {% endif %}
{% endblock %}
```

На примере этой страницы вы без проблем обновите все остальные шаблоны, где выводятся записи.

Задание

Проведите рефакторинг шаблонов проекта и включите в них новый **виджет записи**. Постарайтесь сделать так, чтобы не возникал каскад запросов к базе при обращении к связанным моделям автора и группы. Дополнительно проверьте, что на всех страницах со списком постов установлен **виджет паджинатора**.

Обратите внимание: ваши тесты после рефакторинга не должны сломаться. Запустите их и убедитесь, что всё сделано правильно.

42_Полезные функции

Жизнь слишком коротка, чтобы постоянно рыться в проекте в поиске нужных функций. В Django есть место, где можно найти наиболее востребованные функции: **django.shortcuts**

Расположение модулей, их имена и удобство обращения к ним — это тоже дизайн. Многие вещи в старых версиях Django делались сильно сложнее, чем сейчас.

Часть функций из директории **django.shortcuts** вам уже знакома.

render()

```
render(request, template_name, context=None, content_type=None, status=None, using=None)
```

Вы уже не раз применяли эту функцию для генерации страниц на основе шаблона и списка переменных `context`. В `render()` можно указать MIME-тип отдаваемого документа `content_type`, по умолчанию это `text/html`. Параметр `status` — это код HTTP-ответа сервера; чаще всего используются коды 200 и 404. В параметре `using` можно указать имя движка языка шаблонов, эта нужна на сайтах, где применяется несколько разных движков.

Пример:

```
from django.shortcuts import render

def my_index(request):
    # какой-то код
    return render(request, 'index.html', {'elki': 'palki'},
content_type='text/html', status=200)
```

redirect()

```
redirect(to, *args, permanent=False, **kwargs)
```

Чтобы view-функция ответила переадресацией на другую страницу сайта, применяют **redirect()**. Аргумент `permanent=True` сообщит браузеру и поисковым системам, что редирект постоянный и это надо запомнить.

Редирект может быть полезен при попытке неавторизованного доступа к определённой странице или в случае, если структура адресов сайта была изменена, а пользователь запросил устаревший адрес.

Примеры использования:

```
# models.py
from django.db import models

class MyModel():
    # тут свойства модели
    def get_absolute_url(self):
        return f"/item/{self.id}/"

# views.py
from django.shortcuts import redirect

def my_obj_view(request):
    # ...
    # редирект на объект работает, если в модели есть
    # метод get_absolute_url(), который возвращает путь
    obj = MyModel.objects.get(...)
    return redirect(obj)

def my_name_view(request):
    # ...
    # редирект на страницу по имени, с указанием переменной
    # path("<var>/", views.other, name="path-name"),
    return redirect('path-name', var='any-value')

def my_str_view(request):
    # ...
    # редирект на относительный путь на сайте
    return redirect('/some/url/')

def my_url_view(request):
    # ...
    # редирект на любой другой ресурс в сети
    return redirect('https://example.com/')

get_object_or_404()

get_object_or_404(klass, *args, **kwargs)
```

Функция ищет объект в базе, и если не находит — прерывает работу view-функции и возвращает страницу с ошибкой 404.

Аргумент `klass` — это имя модели, у которой вызывается метод `.get()`.

Слово `class` — это зарезервированное слово в Python, потому даже в описании функций его не применяют. Пришлось выкручиваться и писать `klass`.

Также в аргумент `klass` может быть передан объект `QuerySet` (результат запроса), содержащий один объект.

Именованные аргументы `kwargs` — те же, что используются в методах `get()` и `filter()`.

Пример:

```
from django.shortcuts import get_object_or_404

def my_view(request):
    obj = get_object_or_404(MyModel, pk=1)
```

Без функции `get_object_or_404()` код выглядел бы так:

```
from django.http import Http404

def my_view(request):
    try:
        obj = MyModel.objects.get(pk=1)
    except MyModel.DoesNotExist:
        raise Http404("Объект не найден.")
```

Примеры `get_object_or_404()` с `QuerySet`-объектами:

```
queryset = Post.objects.filter(text__startswith='М')
get_object_or_404(queryset, pk=1)

# то же самое, что и
get_object_or_404(Post, text__startswith='М', pk=1)
```

Если в аргумент `klass` передать `QuerySet` с несколькими объектами, будет вызвано исключение `MultipleObjectsReturned`. То же самое случится, если запрос к модели `get_object_or_404(MyModel, filter=...)` вернёт несколько объектов.

get_list_or_404()

```
get_list_or_404(klass, *args, **kwargs)
```

Функция ищет **список объектов** в базе. Если не находит — прерывает работу view-функции и возвращает страницу с ошибкой 404. Аргументы и принцип работы те же, что и у `get_object_or_404()`, но при получении нескольких объектов не вызывается исключение.

Пример:

```
from django.shortcuts import get_list_or_404

def group_posts(request, slug):
    group = get_object_or_404(Group, slug=slug)
    posts = get_list_or_404(Post, group=group)
    # ...
```

reverse()

```
reverse(viewname, urlconf=None, args=None, kwargs=None, current_app=None)
```

Эта функция позволяет обратиться к view-функции по её имени, указанному в `path()`

В коде эта функция делает то же самое, что и тег `{% url %}` в шаблонах.

Пример:

```
from django.urls import reverse
```

```
def myview(request):
    # для пути
    # path("<username>/<int:post_id>", views.post_view, name="post"),
    path = reverse('post', kwargs={'username': 'leo', 'post_id': 42})
    # или
    path = reverse('post', args=('leo', 42))

    return HttpResponseRedirect(path)
```

43_Оптимизация и кеширование

Оптимизация — это настройка проекта, нацеленная на уменьшение нагрузки и ускорение работы. Универсальных критериев нет, они определяются, исходя из специфики сервиса.

Лучше всего запланировать оптимизацию на финальный этап работы: то, что нам кажется важным в процессе разработки, может оказаться непринципиальным в процессе эксплуатации или решаться совсем другими средствами.

Мозг человека довольно ленив и стремится заниматься простыми или придуманными задачами вместо того, чтобы решать сложные, скучные, но важные задачи. Разработчики часто пишут: «мы работаем над высоконагруженным проектом», но чаще всего это означает, что команда придумала себе интересную игру в оптимизацию. «Строить архитектуру высоконагруженного проекта»™ интереснее, чем работать над сложной и скучной задачей по превращению проекта в коммерчески успешный сервис. Это распространенное когнитивное искажение, о нём надо знать, чтобы вовремя заметить это за собой. Хочется чувствовать себя причастным к чему-то великому, и рассказывать «мы сделали проект, оптимизированный под 10 миллионов посещений в день» приятнее, чем «мы сделали каталог товаров и пытались прикрутить к нему корзину, но проект не окупился».

Оптимизация хороша тем, что большой и сложный механизм, в который вложены многие часы работы множества людей, после оптимизации начинает работать быстрее и устойчивее, «взлетает».

Чем плоха преждевременная оптимизация? Представьте, что вы проектируете особый гоночный автомобиль, например, для ралли на Луне. У вас есть общее представление о том, какие части механизма будут необходимы для работы, но вы пока не решили, где их разместить и как скомпоновать. И вот кто-то говорит, что при разработке нужно оптимизировать длину проводов, и теперь вы мучаетесь, размещая блоки механизма как можно ближе один к другому, у вас нет свободы перекомпоновать систему, а кресло пилота вам придётся установить задом наперёд: ведь это позволит укоротить кабели. Задача по оптимизации решена, а проект провален.

Стратегии оптимизации

Оптимизация — это поиск баланса. Для посещаемого новостного сайта важно, чтобы главная страница быстро загружалась и чтобы на ней были все свежие новости. Но в реальности не критично, если добавленная на сайт новость появится на главной не сразу, а минуту спустя (даже если в техзадании было сказано «мгновенное отображение новостей»). Редакция тратит десятки минут на подготовку даже самой «горячей» новости, а пользователь обновляет страницу раз в несколько минут или просматривает ленту событий утром и вечером перед сном. Можно генерировать главную страницу раз в минуту, сохраняя ее в памяти и отдавая пользователям сохранённую версию. Заплатить за быстрый отзыв страницы и уменьшение нагрузки на БД минутной отсрочкой публикации новости — достойная цена.

Когда становится понятно, что конкретно надо оптимизировать — можно применить разные стратегии. И надо понимать, какую цену придется платить.

- Сделать так, чтобы медленная работа делалась быстро. В другом месте что-то будет работать медленнее.
- Сделать так, чтобы медленная работа делалась один раз. Потребуется дополнительное место для хранения результатов работы.
- Сделать так, чтобы медленная работа делалась заранее. Потребуется сложный механизм планирования для запуска подготовки данных, добавятся сложности отладки, а вся эта система может оказаться невостребованной.
- Сделать так, чтобы медленная работа не делалась вообще, если не нужны результаты. Так работают ORM запросы: они не выполняются до тех пор, пока не потребует обратиться к данным. Но при обращении к связанным данным может получиться каскад запросов.

Оптимизация работы сайта

На нашем сайте пока не видно узких мест, потому что у него один-единственный посетитель. Но представим себе, что на сайт обрушилась волна пользователей (совсем скоро так и будет).

Первые шаги по оптимизации нашего проекта должны быть такими:

- Уменьшить количество генераций сложных страниц за счет кеширования.
- Оптимизировать работу с базой с учётом особенностей нашего проекта.
- Использовать специальный веб-сервер для отдачи статики и сжатия содержимого страниц.

Использование кеширования

При каждом обращении к серверу любая страница проекта **Poemnotes** генерируется с нуля: идут обращения к базе, запрашивается шаблон — и пользователю отдаётся страница. В большинстве случаев цена отрисовки страниц не столь высока, чтобы обращать на нее внимание. Но если посещаемость возрастёт, стоит подумать, как сократить расходы на генерацию HTML.

Представим себе, что к главной странице обращаются 100 раз в секунду, 24 часа в сутки. Арифметика показывает, что за сутки страницу запросят $24*60*60*100 = 8\ 640\ 000$ раз. Если после всех оптимизаций и применения **select_related** и **prefetch_related** для генерации главной страницы будет нужно сделать всего пару десятков запросов к БД, то за сутки к базе будет сделано несколько десятков миллионов однотипных запросов с почти одинаковым результатом.

Если мы будем генерировать главную страницу не при каждом запросе каждого пользователя, а лишь один раз в 15 секунд и станем отдавать сохранённую страницу из памяти, то к базе мы будем обращаться не 100 раз в секунду, а один раз в 15 секунд. Профит: в 1500 раз выгоднее, дёшево и без потерь.

Временное сохранение результатов с целью повторного использования и называется **кеширование**.

Фреймворк кеширования Django

В Django есть встроенный кеширующий бэкенд **locmem.LocMemCache**. Он проверяет, закешированы ли запрошенные данные в памяти или специализированной базе данных.

Если кеша нет — генерирует данные (это может быть результат работы view-функции или даже часть шаблона) и отдает пользователю, одновременно сохраняя их на определённое время в кеш.

Если данные уже были закешированы, то бэкенд проверит, что они не устарели и отдаст их пользователю в готовом виде. Если устарели — заново сгенерирует, отдаст пользователю и перезапишет в кеш.

Для подключения бэкенда кеширования добавьте в `settings.py` следующие строки:

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',  
    }  
}
```

Этот бэкенд годится для разработки, а на боевом сервере обычно используют [Memcached](#) или [Redis](#).

Теперь для кеширования работы view-функции можно применить специальный декоратор:

```
from django.views.decorators.cache import cache_page  
  
@cache_page(60 * 15)  
def my_view(request):  
    ...
```

Число в аргументе указывает, сколько секунд надо хранить значение в кеше.

Включить кеширование можно и в шаблоне, при этом можно кешировать не весь шаблон, а только его часть:

```
{% load cache %}  
{% cache 500 sidebar %}  
    .. колонка сайта ..  
{% endcache %}
```

В первом параметре указывают продолжительность кеширования, во втором — имя ключа, под которым данные хранятся в кеше. Есть и необязательные параметры, они позволяют создавать составной ключ. Например, можно сохранить в памяти сложную часть страницы для конкретного пользователя:

```
{% load cache %}  
{% cache 500 sidebar request.user.username %}  
    .. колонка сайта для залогиненного пользователя ..  
{% endcache %}
```

Оптимизация базы данных

Поиск данных в большой таблице БД может занимать много времени. Для решения этой проблемы были придуманы **индексы**, своего рода справочники, предметные указатели или оглавления. Система прочитывает таблицу и составляет указатель, в котором хранится информация, где искать нужную запись. Индексы создаются не для таблицы целиком, а для столбцов таблиц.

Добавление индексов существенно увеличивает «стоимость» добавления новых записей в базу: после добавления или изменения записей надо будет обновить индекс. Процесс ресурсоёмкий, но это разумная плата за быстродействие системы. Чтение данных в приоритете, ведь посетители чаще читают, чем пишут, и выгоднее редко тратить ресурсы на составление индекса, чем часто — на длительный поиск по БД.

Для оптимизации базы данных надо проанализировать, как происходит поиск записей в моделях. Обычно это обращение по первичному ключу и фильтрация по полям, где указаны связи между таблицами. Некоторые базы автоматически добавляют индексы для внешних ключей. Но так происходит не всегда, и лучше указывать явно, что для этих колонок надо создать индексы. Также есть смысл добавить индексы для полей, по которым часто проводится поиск или сортировка.

Может показаться, что чем больше индексов — тем лучше, но это не так. Лишние индексы в таблице занимают много места на диске. Индексы бывают составными, то есть один индекс может строиться по объединенным данным из нескольких столбцов таблицы, и тогда количество возможных индексов будет равно факториалу количества столбцов, что потребует бесконечного дискового пространства. Кроме того, лишние индексы затормозят работу базы, если планировщик запроса ошибётся в построении плана запроса.

За добавление индекса отвечает аргумент `db_index` в свойствах модели. Для `ForeignKey` это поле по умолчанию имеет значение `True`. Поля модели, для которых указан параметр `unique = True`, тоже имеют индекс по

умолчанию: он нужен при проверке уникальности переданного поля для работы самой базы данных.

Если бы в базе **Poemnotes** были миллионы записей, небольшую оптимизацию дало бы добавление индекса к свойству `pub_date` в модели **Post**:

```
class Post(models.Model):
    # ...
    pub_date = models.DateTimeField("Дата публикации", auto_now_add=True,
db_index=True)
    # ...
```

Для оптимизации поиска по тексту записей можно настроить полнотекстовый индекс базы данных или подключить специализированную базу, ориентированную на работу с текстовыми данными.

Задание

Измените работу главной страницы сайта так, чтобы список записей хранился в кэше и обновлялся раз в 20 секунд. В качестве ключа используйте `index_page`. Напишите тесты, которые проверяют работу кэша.

44_Бонус: установка django-debug-toolbar

Чем сложнее проект, тем больше вопросов: как это работает и откуда вдруг взялась ошибка. У врачей есть УЗИ, рентген и томограф; зрение Супермена позволяет видеть вещи насквозь. У программиста есть тесты и страница ошибок, но если ошибки нет, а сайт ведет себя не так, как ожидалось — хочется обзавестись молотом Тора.

В Django супероружие — это [django-debug-toolbar](#) (сокращенно **DjDT**). Установите его с помощью `pip`:

```
(venv) $ pip install django-debug-toolbar
```

Зарегистрируйте в `settings.py` новое приложение:

```
# Инструмент будет работать только в режиме разработки:
DEBUG = True

# Добавьте новое приложение в
INSTALLED_APPS = [
    # Это приложение необходимо для работы
    "django.contrib.staticfiles",
```

```

# ...
"debug_toolbar",
]

# Добавьте новое приложения для обработки запросов

MIDDLEWARE = [
    # ...
    "debug_toolbar.middleware.DebugToolbarMiddleware",
]

# Добавьте IP адреса при обращении с которых будет доступен инструмент

INTERNAL_IPS = [
    "127.0.0.1",
]

```

В головной файл `urls.py` добавьте новое правило для режима отладки:

```

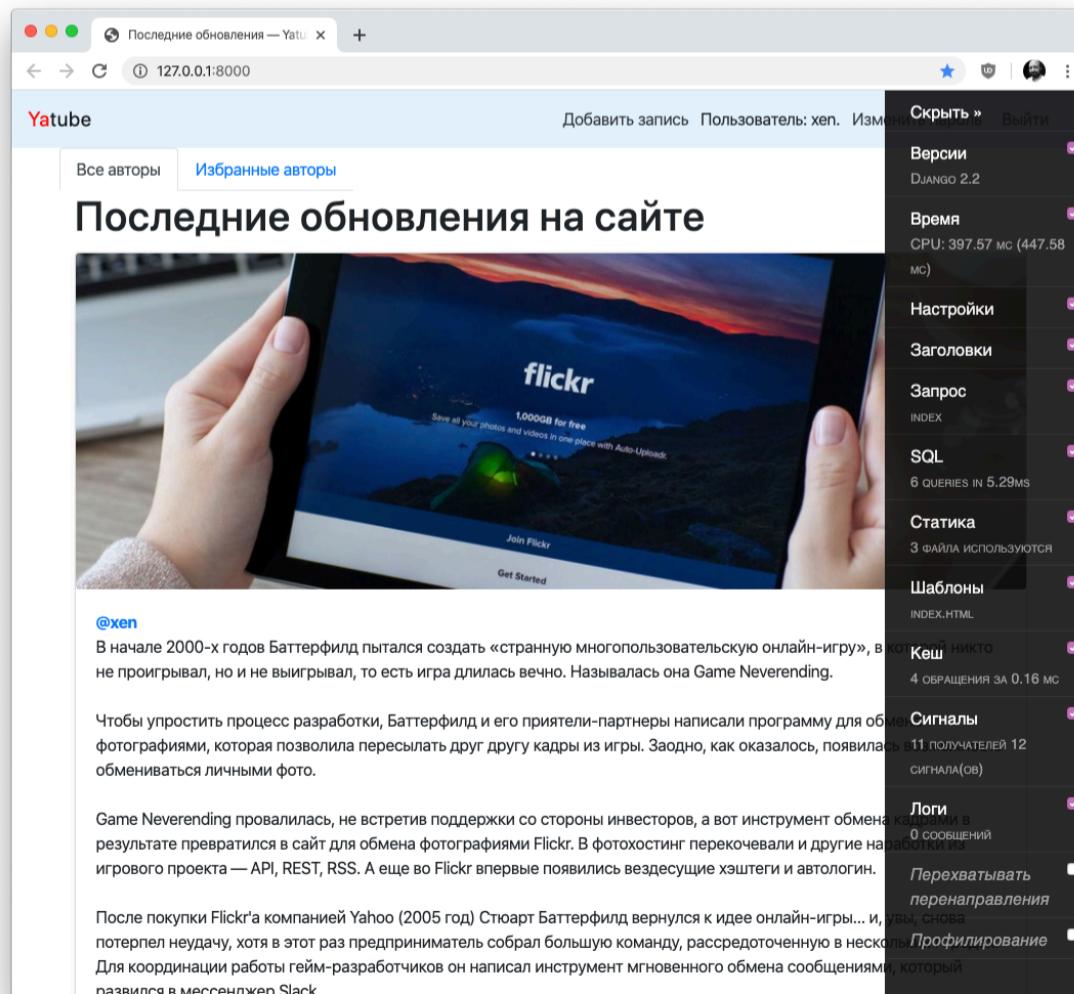
from django.conf import settings

# У вас уже есть это условие
if settings.DEBUG:
    import debug_toolbar

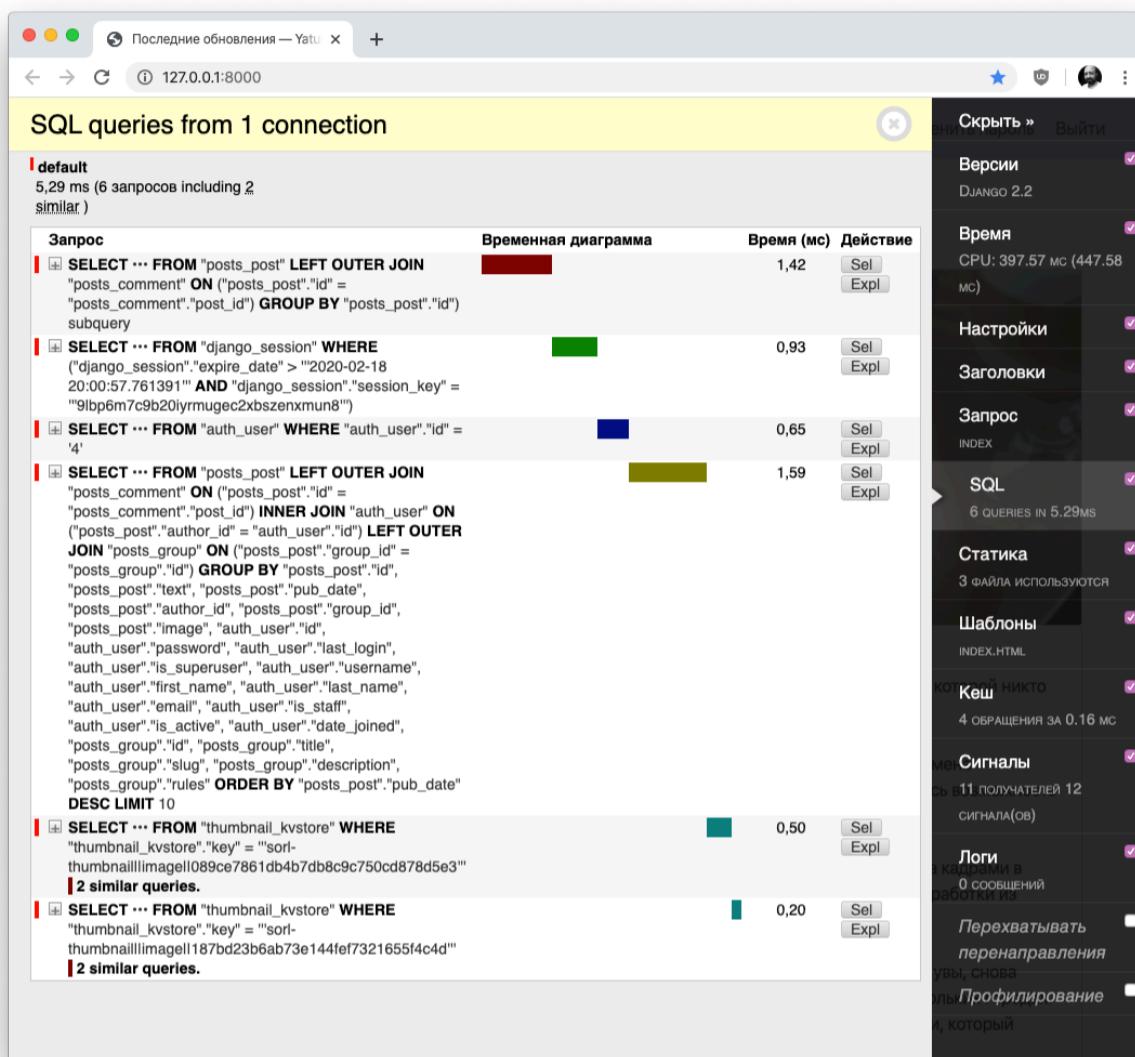
    urlpatterns += (path("__debug__/", include(debug_toolbar.urls)),)

```

Откройте сайт — и у вас появится новая панель:



Теперь вам доступна груда информации о том, как шла отрисовка текущей страницы. Например, вот список запросов, которые выполнялись перед отображением главной страницы:



Каждый запрос можно рассмотреть подробнее (кнопка *Sel*) или узнать, какие индексы и какой план запроса был сформирован для получения данных (кнопка *Expl*, от *explain*).

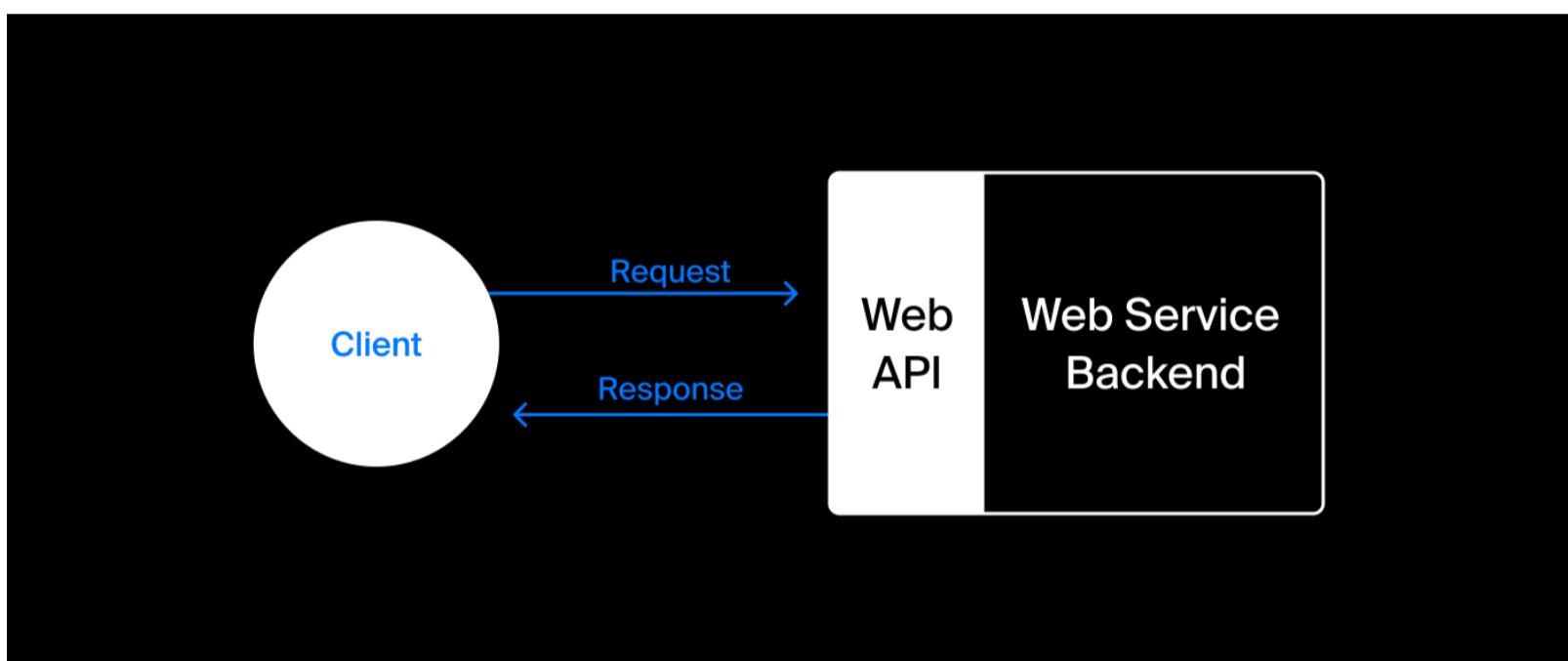
Большинство Django-программистов подключают **DjDT** к своим проектам по умолчанию, это упрощает разработку и помогает избежать многих проблем.

45_Что такое API. Формат JSON

Интерфейс (от англ. *interface*, «взаимодействие») — это средство взаимодействия со сложной системой. Регулятор громкости радиоприёмника — это интерфейс регулирования громкости, посредник между человеком и электронной системой.

Другой пример: графический интерфейс операционной системы компьютера. Мы решаем свои повседневные задачи, не задумываясь о внутреннем устройстве самой операционной системы. При перетаскивании иконки файла в «корзину» файл удаляется, в файловой системе происходят изменения.

API (от англ. **A**pplication **P**rogramming **I**nterface, «программный интерфейс приложения») — это интерфейс для обмена данными. Слово «программный» означает, что интерфейс спроектирован в первую очередь для программ, и в этом смысле с системой взаимодействует не разработчик, а код, написанный им.



Вы уже работали с API во вводном курсе: Анфиса отправляла запрос к API и получала информацию о погоде на завтра.

API могут общаться друг с другом: например, ваш код запрашивает данные с API Яндекс.Маркета о появлении новых товаров, а Яндекс.Маркет в свою очередь обращается к API различных магазинов, чтобы эти данные получить. При этом каждый сервис может быть написан на своем языке программирования, но именно благодаря API они могут легко общаться между собой по сети, используя протокол HTTP и передавая данные в удобном для всех формате. Один из самых распространенных форматов передачи данных — JSON, он пришёл на смену формату XML.

JSON расшифровывается как ***J*ava*S*cript *O*bject *N*otation** (англ. «объектная запись JavaScript»). Так сложилось исторически: этот формат «вырос» из языка программирования *JavaScript* и он очень похож на тип данных ***dict***, но более стандартизирован. Например, **ключи нужно записывать только в**

двойных кавычках — в JSON это обязательно. Значениями ключей могут быть строки, числа, булевые значения, словари и списки. Некоторые типы данных не поддерживаются, например тип даты ("09-10-1988") записывается как строка.

Больше о формате JSON можно почитать [в документации](#)

Чтобы убедиться в том, что JSON правильно составлен, можно обратиться к любому онлайн-валидатору, например [вот к этому](#)

Вот как выглядят данные из картотеки супергероев, записанные в формате JSON:

```
[  
  {  
    "name": "Captain America",  
    "realName": "Steve Rogers",  
    "yearCreated": 1941,  
    "powers": [  
      "Strength",  
      "Healing ability"  
    ]  
  },  
  {  
    "name": "Spider-Man",  
    "realName": "Peter Parker",  
    "yearCreated": 1963,  
    "powers": [  
      "Danger sense",  
      "Speed",  
      "Jumping"  
    ]  
  }  
]
```

А вот пример этих же данных в формате XML (внешне этот язык чем-то похож на HTML, и это неслучайно: язык разметки веб-страниц — прямой потомок XML). Названия тегов в XML не стандартизированы, их можно называть по собственному разумению.

```
<?xml version="1.0" encoding="UTF-8" ?>  
<root>  
  <superhero>  
    <name>Captain America</name>  
    <realName>Steve Rogers</realName>  
    <yearCreated>1941</yearCreated>  
    <powers>Strength</powers>  
    <powers>Healing ability</powers>  
  </superhero>  
  <superhero>
```

```
<name>Spider-Man</name>
<realName>Peter Parker</realName>
<yearCreated>1963</yearCreated>
<powers>Danger sense</powers>
<powers>Speed</powers>
<powers>Jumping</powers>
</superhero>
</root>
```

Проверить валидность XML можно с помощью любого онлайн-валидатора, например, через [валидатор на Codebeautify.org](#)

Более подробно об XML можно почитать в документации

JSON — один из наиболее популярных форматов обмена данными при работе с API, мы с ним будем много работать. Он проще для чтения и чуть меньше по размеру, чем такие же данные в XML.

46_API First. Архитектура REST

API сервисы раньше и сейчас

Пока не было мессенджеров и смартфонов — в интернет выходили с настольного компьютера, а единственным пользовательским клиентом был веб-браузер. Сайты работали так: сервер получал запрос и отдавал клиенту готовые HTML-страницы.

В современном мире у такого подхода есть два недостатка:

- HTML-страница нужна браузеру, но не мобильному приложению — данные внутри него организованы иначе. Но если у какого-то сервиса есть и сайт, и мобильное приложение, они будут общаться с одним сервером. Поэтому разработчику придётся делать для мобильного приложения отдельный способ получения данных.
- Поскольку HTML-код генерируется на сервере, при переходе от страницы к странице браузер перезагрузит сайт целиком. Это неэффективно, поскольку приходится перерисовывать одинаковые элементы: шапку, меню, подвал. Разумнее получить данные об изменяющихся частях сайта и отрисовать только их.

Размышляя об этих проблемах, Тим Бёрнерс-Ли (которого можно назвать создателем интернета) и Рой Филдинг (его коллега) придумали принципы, которые позволяли бы масштабировать развитие всемирной сети.

Основная идея в том, что с сервера возвращают только данные, а клиент уже сам разбирается, как эти данные отрисовать. Мобильное приложение будет использовать свои методы отрисовки, браузер или чат-бот — свои. Так мы ограничиваемся одним API для разных платформ. Такой подход получил название **API-First**, то есть *сначала данные, а затем — интерфейсы для их отображения*.

Так появился **REST**.

REST

REST, или REpresentational State Transfer (англ. «передача состояния представления») — это набор принципов, которых следует придерживаться при создании API. Если API сделан по этим принципам, его называют **RESTful API** (или просто **REST API**).

Эти принципы стандартизируют передачу данных по сети, они похожи на правила вежливости, когда людям (серверам) удобно находиться в одном обществе (интернете) и понимать друг друга.

Принципы REST

Эти принципы ввёл Рой Филдинг (*Roy Fielding*) в 2000 году, в своей диссертации «Архитектурные стили и дизайн сетевых программных структур».

1. Клиент-сервер. Разделение ответственности между клиентом и сервером

Клиент и сервер отвечают за разные вещи. Ответственность клиента — пользовательский интерфейс, ответственность сервера — данные. Если API возвращает HTML-страницу, его нельзя назвать REST API: ведь при этом сервер берёт на себя ответственность за интерфейс.

2. Отсутствие состояния. Сервер не хранит состояние

Каждый запрос должен быть независимым, как будто он сделан в первый раз. Сервер не должен хранить какой-либо информации о клиенте. Запрос клиента к серверу должен содержать всю информацию, необходимую для обработки этого запроса: кто запрашивает данные, какие данные запрашиваются.

3. Единый интерфейс

Интерфейс обращения к серверу одинаков для всех и не зависит от клиента. Запрос к данным может быть сформирован из браузера, мобильного приложения и с умного чайника по одним и тем же правилам.

4. Многоуровневость

Первый принцип гласит, что в коммуникации участвуют двое: клиент и сервер. Но можно строить более сложные системы, не нарушая этого принципа.

API сервиса Яндекс.Такси может использовать API Яндекс.Навигатора. Вы как клиент взаимодействуете только с API Яндекс.Такси, а он в свою очередь является клиентом навигатора. Здесь есть одно условие — каждый компонент должен видеть только свой уровень, например, Яндекс.Навигатор не должен видеть все данные, которые вы отправили в Яндекс.Такси.

5. Кэшируемость

Данные ответа могут быть закэшированы. Это значит, что можно сохранить данные на клиенте, а при идентичном запросе взять их из памяти клиента — кэша, а не ждать их с сервера. Нет смысла запрашивать данные повторно, если они никак не изменились.

6. Код по запросу

Этот принцип необязательный. Он гласит, что функциональность клиента может быть расширена кодом, приходящим с сервера. Сейчас такое можно встретить повсеместно: JavaScript используется для «оживления» страниц и исполнения каких-то сценариев на стороне клиента. Но принципы формулировались в 2000 году — тогда исполняемый код с сервера возвращали не так часто. Потому и выделили это в отдельный принцип.

47_Правила именования ресурсов

Ресурс — это единица информации в терминологии REST API. Ресурсом может быть документ, один пост в блоге, или список постов, пользователь социальной сети — всё, к чему можно обратиться при HTTP-запросе. Это концептуальное отображение какого-то объекта или набора объектов, но не

сами эти объекты. Мы отображаем данные в текстовом формате, чтобы выполнять с ними какие-то действия.

JSON и XML — это форматы данных для текстового представления ресурсов.

Правила именования ресурсов

Для обращения к ресурсам через HTTP-запросы их необходимо как-то именовать. Чем понятнее имена ресурсов, тем проще разобраться в API. Здорово, когда по названию ресурса можно понять, **что** он содержит. Чтобы с вашим API было просто и удобно общаться — соблюдайте простые правила.

Ресурсы — существительные

Почти всегда ресурсы именуют существительными во множественном числе:

```
https://swapi.co/api/starships
```

Иногда для именования ресурсов применяют единственное число, но это реже:

```
https://praktikum.ru/users/{user-id}/profile
```

```
https://praktikum.ru/users/me
```

Иногда можно выбрать глагол в качестве имени. Но такое имя бывает удачным крайне редко:

```
# проверка корзины
```

```
# для получения токена
```

```
# для обновления токена
```

Слэш для иерархии

Слэш в URL используется для указания иерархии ресурсов по принципу «от общего к частному»:

```
GET /users/{user-id}/posts
```

Все пользователи → Конкретный пользователь по ID → Все его посты
То есть запрос вернет посты конкретного пользователя. Если мы хотим получить конкретный пост — нужно указать его ID:

```
GET /users/{user-id}/posts/{post-id}
```

«Висячий» слэш — не рекомендуется

Висячий слэш в конце URL не добавляет информации. Лучше избегать такого употребления.

```
// не надо так  
GET /users/{user-id}/posts/
```

```
// лучше вот так  
GET /users/{user-id}/posts
```

Дефисы вместо пробелов

В URL не должно быть пробелов, их заменяют либо дефисами, либо нижними подчёркиваниями. Дефисы лучше, потому что:

- на некоторых устройствах нижнее подчёркивание может выйти за базовую линию строки, и его будет не видно вовсе;
- несколько нижних подчёркиваний сливаются в одно.

```
// делайте так:  
GET /users/{user-id}/user-devices
```

```
// а не так:  
GET /users/{user-id}/user_devices
```

HTTP-методы

Любой API предназначен для получения доступа к ресурсам сервера. К ресурсу всегда можно обратиться по URL.

HTTP-метод запроса определяет, что следует сделать:

- **GET** получает ресурсы;
- **POST** создаёт ресурс;

- **PUT** заменяет существующий ресурс целиком;
- **PATCH** частично изменяет существующий ресурс;
- **DELETE** удаляет ресурс.

Реже применяют ещё два метода:

- **HEAD**, получить только заголовки ответа. HEAD похож на GET, но у его ответа нет тела;
- **OPTIONS**, узнать, какие HTTP-методы поддерживает сервер.

Сам по себе метод не определяет логику работы сервера. Можно закодить API так, чтобы запрос POST удалял ресурс, а DELETE — добавлял.

Поступайте именно так, когда вам поставят задачу «нарушить как можно больше стандартов и усложнить работу коллегам».

HTTP-методы — это глаголы

Используйте HTTP-методы для указания совершаемого над ресурсом действия:

```
// получить список пользователей  
GET /users
```

```
// создать пользователя  
POST /users
```

```
// удалить пользователя с id = 2  
DELETE /users/2
```

А включать в название имя HTTP-метода не стоит:

```
# так не надо  
GET /get-users  
POST /create-user
```

Из названия HTTP-метода уже понятно, что делать с ресурсом, не нужно дублировать задание.

Идемпотентность и безопасность методов

У методов есть две характеристики — безопасность и идемпотентность.

Безопасность: если метод может изменить ресурс, то он считается небезопасным в терминах архитектуры REST. Такими методами могут быть PUT, PATCH, DELETE или POST.

Идемпотентность метода заключается в том, что его многократное повторение равно однократному. То есть выполняя один и тот же запрос много-много раз, мы всегда будем получать один и тот же результат.

48_Механизмы авторизации, протокол OAuth 2.0

Освежим в памяти понятия «авторизация» (англ. *authorization*, «разрешение, одобрение») и «аутентификация» (англ. *authentication*, «опознание»).

Авторизация — предоставление пользователю прав на выполнение каких-то действий.

Аутентификация — процедура проверки подлинности пользователя.

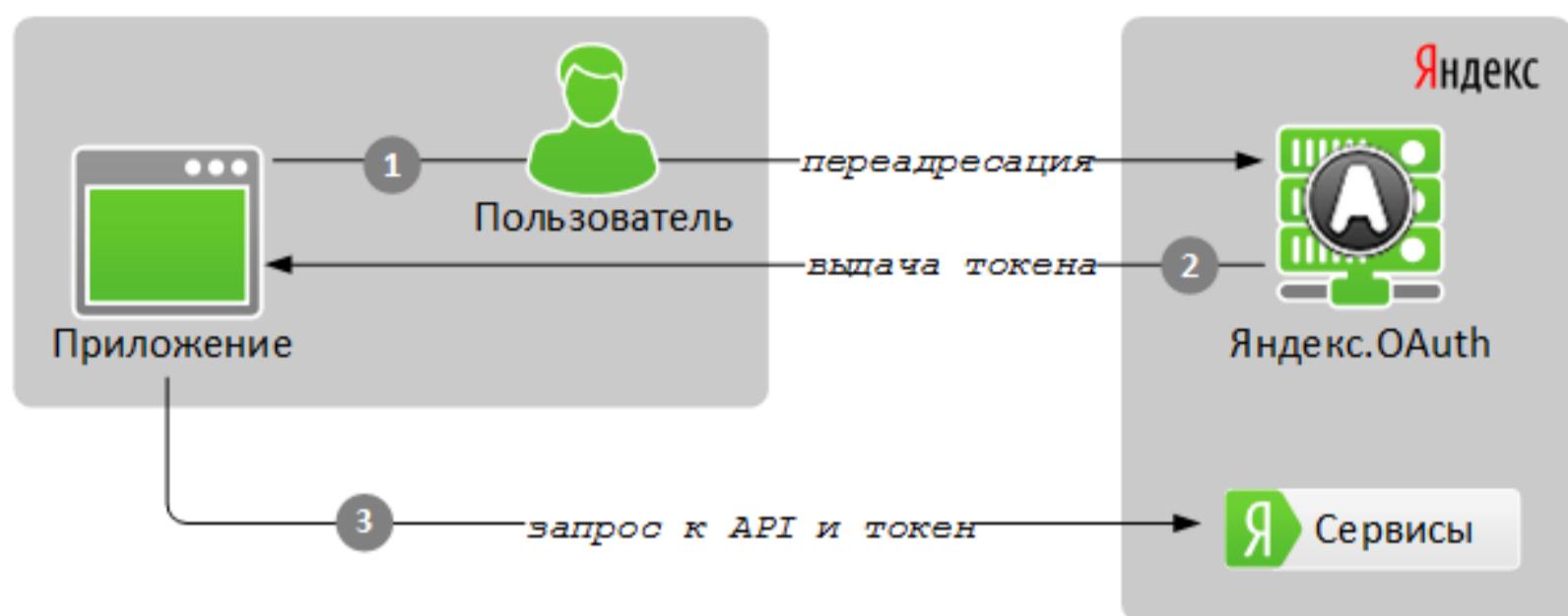
Представьте, что вы поехали отдыхать и вам нужна виза. **Аутентификацию** можно сравнить с получением этой визы, процессом её изготовления.

Допустим, вы успешно получили визу (прошли процесс **аутентификации**) и прилетели на отдых: пограничник проверит вашу визу, сравнит фотографию и паспортные данные — выполнит процедуру **авторизации** и предоставит право на посещение страны. И каждый раз, когда вы вновь посетите эту страну, вам будет нужно **авторизоваться** на паспортном контроле.

Вы часто проходите авторизацию по логину и паролю: вы входите в систему и получаете доступ к сервису. Но иногда нужно предоставить лишь частичный доступ. Например, вы просите другого человека загрузить фотографии в свой аккаунт социальной сети, но не хотите давать доступ к вашим личным сообщениям. Или вы пишете приложение для автоматической загрузки фотографий в социальную сеть, но не хотите

наделять эту программу полным доступом к своей учетной записи. Тут на помощь приходит протокол авторизации OAuth.

OAuth (Open Authorization) — это схема авторизации, предоставляющая третьей стороне (другому пользователю или приложению) ограниченный доступ к ресурсам сервиса от вашего имени, без необходимости передавать логин и пароль. Это становится возможным благодаря **OAuth-токену**.



Токен (англ. "token" – опознавательный знак, жетон) — это уникальная строка из цифр и латинских букв, он может выглядеть так:

```
34bfedf5e4c5e8858b9ebcb4821e12cd806ac4c93e3b50d5adbdae2b2
```

OAuth-токен выдается пользователю для упрощения доступа к серверу. Это что-то вроде пропуска для входа на определённый ресурс. Такой пропуск может быть временным — выдаваться на какой-то срок, или пожизненным.

В обычной жизни вы, например, можете выдать своему другу «токен» с ограниченными правами — доверенность на машину. Друг сможет управлять вашей машиной, но у него не будет права её продать.

При авторизации достаточно лишь «показать пропуск» — передать токен вместе с запросом на сервер. По этому токену сервер поймёт, кто к нему обращается и к каким данным владелец токена имеет доступ.

Если злоумышленник завладеет вашим токеном (найдёт на улице ваш пропуск), то он получит доступ к данным и сможет совершать запросы от вашего имени; крайне важно хранить токен в секрете и никому его не

сообщать. А если беда всё же случилась — токен можно отозвать в любой момент, это удобно и добавляет безопасности.

49_Получение токена ВКонтакте

Рассмотрим авторизацию по протоколу OAuth 2.0 на примере доступа к API популярной социальной сети «ВКонтакте».

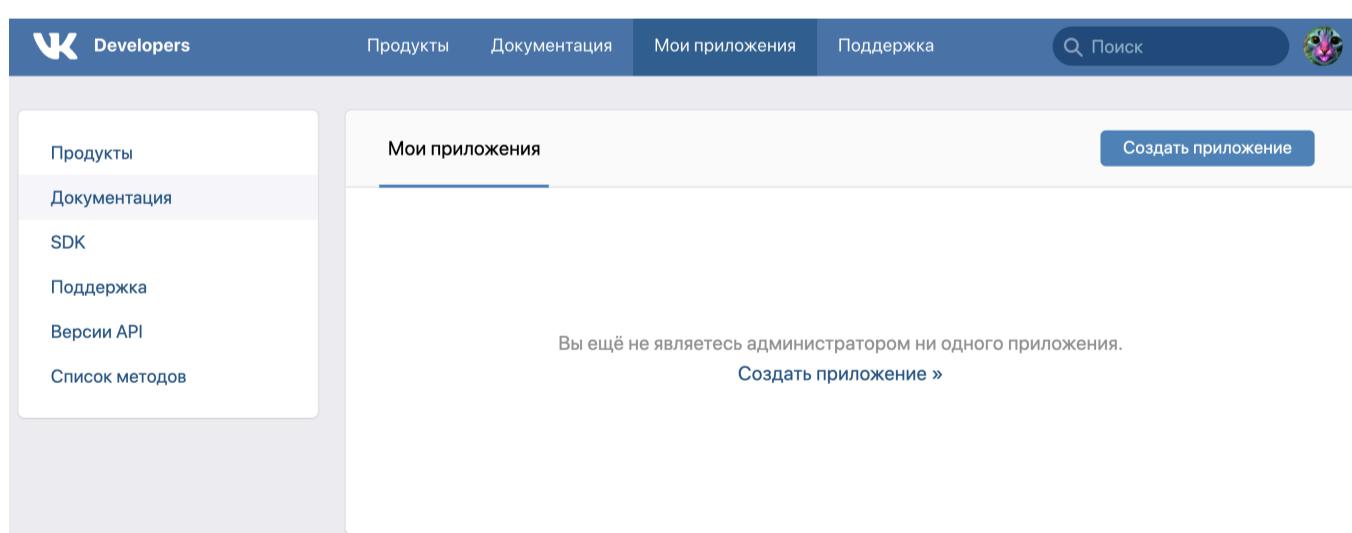
Авторизация

Войдите под своим логином и паролем в сервис ВКонтакте. Если у вас там нет аккаунта, то для решения задач создайте учебный профиль. Он потребуется вам для выполнения дальнейших практических заданий.

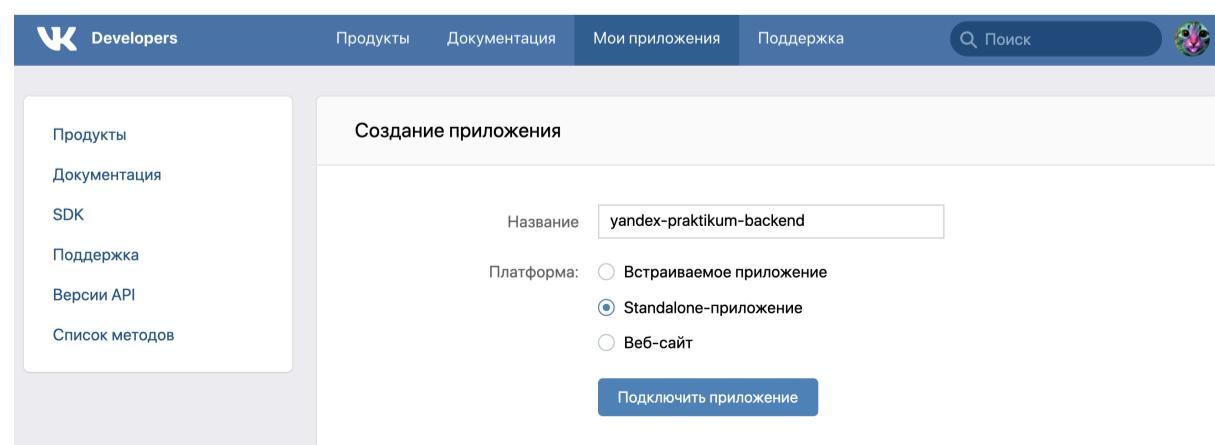
Создание приложения

На специальной [страничке для разработчиков](#) зайдите во вкладку «Мои приложения». Нажмите на кнопку «Создать приложение» и дайте приложению понятное название, затем выберите «Standalone-приложение» как платформу. Так в терминах протокола OAuth 2.0 называется стороннее приложение (ваша программа, которую вы напишете), совершающее запросы от вашего имени.

В результате приложение появится в списке "Мои приложения".



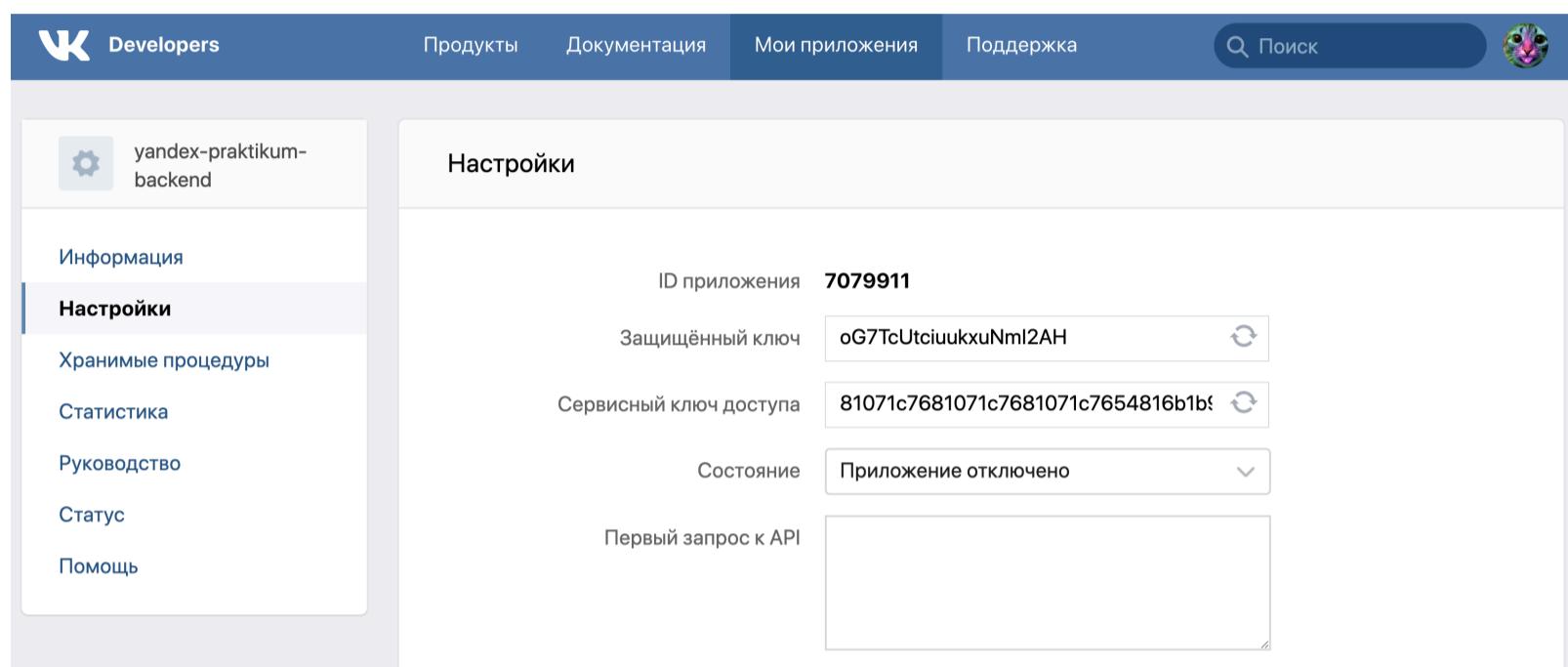
The screenshot shows the VK Developers website interface. At the top, there is a navigation bar with links for 'Products', 'Documentation', 'My applications' (which is currently selected and highlighted in blue), and 'Support'. There is also a search bar and a user profile icon. On the left side, there is a sidebar with links for 'Products', 'Documentation', 'SDK', 'Support', 'API Versions', and 'List of methods'. The main content area is titled 'My applications' and contains the message: 'You are not yet an administrator of any application.' with a link 'Create application >'. A blue button labeled 'Create application' is located at the top right of this section.



The screenshot shows the 'Create application' form on the VK Developers website. The form has a title 'Create application'. It contains fields for 'Name' (with the value 'yandex-praktikum-backend') and 'Platform' (with the selected option 'Standalone-application'). Below the platform selection, there are two other options: 'Embedded application' and 'Website'. At the bottom of the form is a blue button labeled 'Connect application'.

Скопируйте идентификатор (ID) приложения

Найдите только что созданное приложение в списке «Мои приложения» и нажмите «Редактировать» рядом с его названием. Затем перейдите во вкладку «Настройки» слева на экране и скопируйте идентификатор из строки «ID приложения» — он вам пригодится.



The screenshot shows the VK Developers application settings interface. The top navigation bar includes links for 'Developers', 'Продукты' (Products), 'Документация' (Documentation), 'Мои приложения' (My Applications), 'Поддержка' (Support), and a search bar. A user profile icon is in the top right. On the left, a sidebar menu lists 'Информация', 'Настройки' (selected), 'Хранимые процедуры', 'Статистика', 'Руководство', 'Статус', and 'Помощь'. The main content area is titled 'Настройки' and displays the application 'yandex-praktikum-backend'. Key settings shown include:

- ID приложения: 7079911
- Защищённый ключ: oG7TcUtcuuukxuNmI2AH (with a refresh icon)
- Сервисный ключ доступа: 81071c7681071c7681071c7654816b1b (with a refresh icon)
- Состояние: Приложение отключено (with a dropdown arrow)
- Первый запрос к API: (empty text area)

Шаг 4.

Скопируйте в адресную строку браузера запрос, который вернёт вам токен для доступа к данным вашего аккаунта. Полученный на прошлом шаге идентификатор приложения передайте в параметре *client_id*:

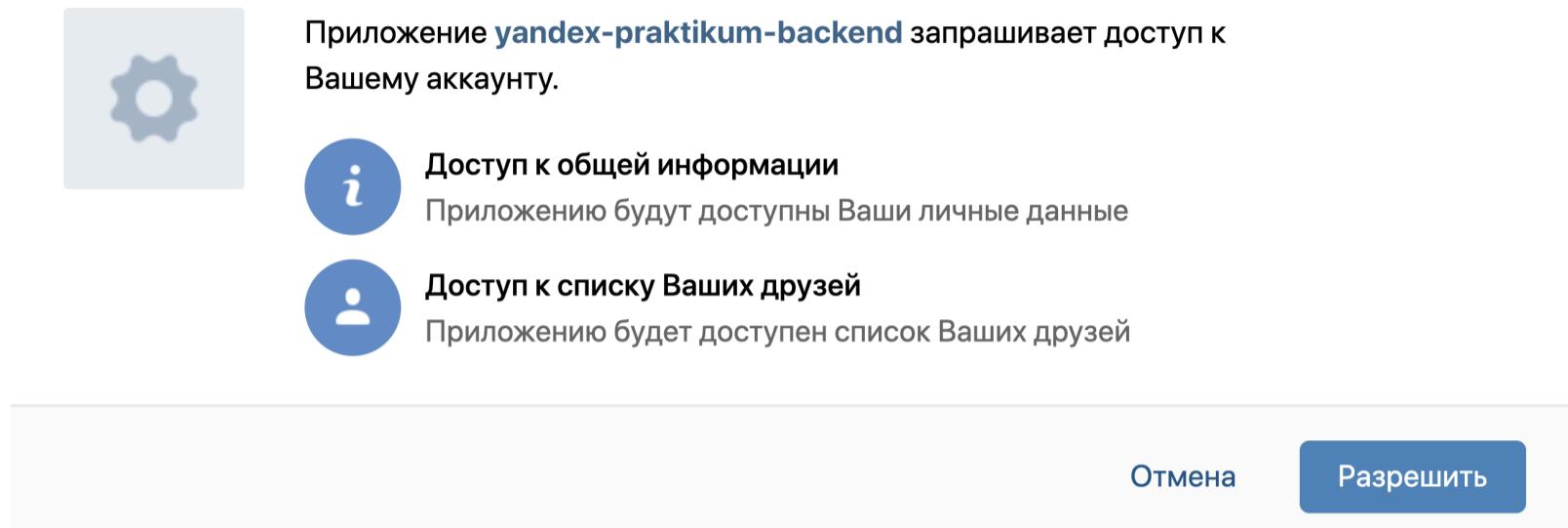
```
https://oauth.vk.com/authorize?  
client_id=XXXXXX&scope=friends&response_type=token&v=5.92
```

Эту ссылку можно получить в любой момент в кабинете разработчика ВКонтакте.

В параметре *scope* указывается, к каким данным аккаунта будет разрешён доступ. Значение *scope=friends* даёт владельцу токена право управлять вашими друзьями (друзьями пользователя того аккаунта, от имени которого получен токен).



Откроется страница, где сервис ВКонтакте запросит у вас разрешение на доступ по токену к тем ресурсам, которые вы перечислили в параметре `scope`:



После получения разрешения вас перенаправят на другую страницу, а в адресной строке появится параметр `access_token`



`access_token` — это и есть ваш токен. Сохраните его.

`expires_in` — время жизни токена в секундах.

`user_id` — ID пользователя, которому принадлежит токен.

В целях безопасности время жизни токена ограничено сутками, он будет действовать 86400 секунд. По истечении этого времени ваш токен превратится в тыкву — и придётся получать новый.

Можно получить и вечный токен, для этого в запросе на получение токена в параметр `scope` нужно добавить значение `offline`:

```
https://oauth.vk.com/authorize?  
scope=friends,offline&response_type=token&v=5.92&client_id=XXXXXX
```

50_Как хранить секреты

Никогда не храните секретные данные (логины, пароли, токены, ключи доступа) в вашем коде, это плохая практика. В функции и в объекты любой ключ доступа должен передаваться только в виде переменной, хранящей этот ключ. Тут вам поможет библиотека `dotenv`

```
# небезопасный вариант
# отправляем запрос с токеном к API ВКонтакте,
# токен пишем прямо в функции.
# никогда_так_не_делайте
def any_function():
    secret_token = {'token': 'my_super_secret_token'} # шпионы рады, шпионы
    # крадут токен отсюда
    response_post = requests.post('https://api.vk.com/method/users.get',
data=token)

# безопасный вариант
# отправляем запрос с токеном к API ВКонтакте
# токен получаем из специального хранилища
# = только так и делайте =
from dotenv import load_dotenv
load_dotenv()

def other_function():
    secret_token = '123'
    # взяли переменную token из загруженного внешнего хранилища
    # шпионы печальны, шпионы ушли с пустыми руками
    response_post = requests.post('https://api.vk.com/method/users.get',
data=secret_token)
```

В следующих уроках вы будете много работать с токенами, не пренебрегайте правилами безопасности даже во время учёбы, пусть лёгкая паранойя войдёт в привычку и станет рефлекторной.

Когда в коде используются какие-то секретные данные, импортируйте их в код извне. Это можно сделать как минимум двумя способами:

Локальное хранение секретных ключей

Создайте файл `.env` (все файлы, которые начинаются с точки — скрыты от просмотра в консоли и в некоторых операционных системах) и запишите в него переменные в формате `ключ=значение`, по одной переменной на строку:

```
Token=123
Login=test
Sid=some_sid
```

Для доступа к переменным установите через `pip` библиотеку `dotenv`:

```
pip install python-dotenv
```

Затем импортируйте и выполните функцию `load_dotenv`:

```
import os
from dotenv import load_dotenv
load_dotenv()

token = os.getenv('Token')
print(token) # 123
```

Файл `.env` должен лежать в той же директории, что и исполняемый файл.

Никогда не храните файл `.env` в репозитории! Проверьте, что в `.gitignore` есть строчка `.env`, чтобы случайно не запушить файл на удалённый сервер.

Глобальное хранение секретных ключей

Выполните в баше команду `export token=123`, это запишет токен в общее пространство переменных окружения. Просмотреть содержимое этого пространства можно командой `env` в терминале.

Чтобы получить доступ к переменным окружения из кода, воспользуйтесь методом `getenv` из встроенного модуля `os`:

```
import os

account_sid = os.getenv("account_sid")
auth_token = os.getenv("token")
```

Для доступа к глобальным переменным достаточно метода `os.getenv`.

51_Запрос к API сервиса ВКонтакте

Самое время подёргать за ниточки API ВКонтакте и посмотреть, что мы можем получить. Посмотрите, что интересного есть на [странице документации API ВК](#).

Методами API здесь называют команды, выполняющие запросы к серверу ВКонтакте. Например, метод `photos.add` добавляет фотографию, а метод `gifts.get` возвращает список подарков определенного пользователя.

Место для того, чтобы открыть страшную правду: Вконтакте не соблюдает REST. Mail.ru, Twitter и другие крупные сервисы тоже так делают: многие крупные компании разрабатывают собственные внутренние стандарты, и с этим приходится мириться. \(\times)/

Методы разделены на группы: для работы с друзьями пользователя используют методы **friends**, для работы с лайками — **likes**. У метода могут быть **параметры**: это какие-то входные данные (например, `online=1`). Перейдите на [страницу «Список методов»](#), в навигации ВК эта ссылка в самом низу слева.

The screenshot shows the VK Developers API documentation interface. At the top, there's a navigation bar with links for 'Продукты', 'Документация', 'Мои приложения', 'Поддержка', a search bar, and a user profile icon. On the left, a sidebar lists various API categories like 'Продукты', 'Документация', 'SDK', 'Поддержка', 'Версии API', and 'Список методов'. The 'Список методов' link is highlighted with a blue border. The main content area is titled 'Описание методов API' and contains a note: 'Ниже приводятся все методы для работы с данными ВКонтакте.' Below this, the 'Account' category is expanded, showing a table of methods:

| Метод | Описание |
|--|--|
| <code>account.ban</code> | Добавляет пользователя или группу в черный список. |
| <code>account.changePassword</code> | Позволяет сменить пароль пользователя после успешного восстановления доступа к аккаунту через СМС, используя метод <code>auth.restore</code> . |
| <code>account.getActiveOffers</code> | Возвращает список активных рекламных предложений (офферов), выполнив которые пользователь сможет получить соответствующее количество голосов на свой счет внутри приложения. |
| <code>account.getAppPermissions</code> | Получает настройки текущего пользователя в данном приложении. |
| <code>account.getBanned</code> | Возвращает список пользователей, находящихся в черном списке. |
| <code>account.getCounters</code> | Возвращает ненулевые значения счетчиков пользователя. |
| <code>account.getInfo</code> | Возвращает информацию о текущем аккаунте. |
| <code>account.getProfileInfo</code> | Возвращает информацию о текущем профиле. |
| <code>account.getPushSettings</code> | Позволяет получать настройки Push-уведомлений. |
| <code>account.registerDevice</code> | Подписывает устройство на базе iOS, Android, Windows Phone или Mac на получение Push-уведомлений. |
| <code>account.saveProfileInfo</code> | Редактирует информацию текущего профиля. |
| <code>account.setInfo</code> | Позволяет редактировать информацию о текущем аккаунте. |
| <code>account.setNameInMenu</code> | Устанавливает короткое название приложения (до 17 символов), которое выводится пользователю в левом меню. |
| <code>account.setOffline</code> | Помечает текущего пользователя как offline (только в текущем приложении). |
| <code>account.setOnline</code> | Помечает текущего пользователя как online на 5 минут. |
| <code>account.setPushSettings</code> | Изменяет настройку Push-уведомлений. |
| <code>account.setSilenceMode</code> | Отключает push-уведомления на заданный промежуток времени. |
| <code>account.unban</code> | Удаляет пользователя или группу из черного списка. |
| <code>account.unregisterDevice</code> | Отписывает устройство от Push уведомлений. |

Давайте добудем из ВК данные о каком-нибудь пользователе.

Список методов для работы с пользователями собран под вкладкой *Users*. Необходимый метод API называется `users.get`, он принимает на вход необязательный параметр `user_ids`, идентификатор пользователя, один или несколько. Если `user_ids` не указан явно — API вернёт данные владельца токена.

Соберём нужный запрос:

```
https://api.vk.com/method/users.get?user_id=&v=5.92&access_token=XXX
```

... где вместо `xxx` — ваш токен.

Рассмотрим этот запрос подробнее.

https:// — указывает на то, что мы осуществляляем запрос через протокол HTTPS, *HyperText Transfer Protocol Secure* (англ. «защищённый HTTP»).

api.vk.com/method — адрес сервиса API ВКонтакте.

users.get — название метода API.

После вопросительного знака идут параметры GET-запроса, разделенные символом `&`. Так мы передаём методу входные данные.

В нашем запросе три параметра:

user_ids= необязательный параметр метода, указывает на пользователя или список пользователей.

v=5.92 указывает на то, что формат данных должен соответствовать версии API равной 5.92 (обычно нужно указывать последнюю версию).

access_token= токен авторизации, вы получили его ранее.

В ответ на ваш запрос сервер вернет **JSON-объект** с данными. Например, для пользователя с идентификатором 210700286:

```
{  
    "response": [  
        {  
            "id": 210700286,  
            "first_name": "Lindsey",  
            "last_name": "Stirling",  
            "is_closed": false,  
            "can_access_closed": true  
        }  
    ]  
}
```

```
← → C https://api.vk.com/method/users.get?user_ids=210700286&v=5.92&access_token=66dacdc855aaaa6428d8a647e104f8a78045  
{"response": [{"id": 210700286, "first_name": "Lindsey", "last_name": "Stirling", "is_closed": false, "can_access_closed": true}]}
```

Если в запросе допущена ошибка, например, не передан `access_token`, сервер вернет сообщение об ошибке, с кодом и описанием:

```
← → C https://api.vk.com/method/users.get?user_ids=210700286&v=5.92&access_token=  
{"error": {"error_code": 5, "error_msg": "User authorization failed: no access_token passed.", "request_params": [{"key": "user_ids", "value": "210700286"}]}
```

52_Отправка SMS-уведомлений

Практика SMS-уведомлений от приложений и сайтов весьма популярна и распространена: подтверждение заказа в интернет-магазине, валидация номера телефона, банковские уведомления — в каждом из этих случаев пользователь получает на свой телефон текстовое сообщение.

Многие считают, что эти сообщения рассылают специально обученные SMS-гномики, но на самом деле это не так.

Есть множество сервисов, предоставляющих API для отправки SMS. Настроим отправку сообщений через одну из таких систем и разрушим миф о гномиках.

- Зарегистрируйтесь на Twilio и подтвердите email и телефон. Сервис предоставит вам \$15 для тестирования системы, этого хватит примерно на 300 сообщений.

- Откройте главную страницу работы с Twilio, здесь можно посмотреть всю основную информацию о сервисе: остаток баланса, номера, ключи доступа и всё остальное.

Получите номер, с которого Twilio будет отправлять ваши сообщения. Выберите решетку # в меню слева и нажмите красную кнопку "Get your first Twilio phone number".

The screenshot shows the Twilio web interface. In the top left, there's a sidebar with links like 'Manage Numbers', 'Buy a Number', 'Verified Caller IDs', etc. The main content area has a heading 'Get Started with Phone Numbers'. Below it, there's a paragraph about getting started with Twilio phone numbers, followed by a prominent red button labeled 'Get your first Twilio phone number'. Further down, there's information about alphanumeric sender IDs and helpful documentation, along with another red button for upgrading the account.

Вам предложат номер телефона. Снова нажмите красную кнопку "Choose this Number":

This screenshot shows a modal window titled 'Your first Twilio Phone Number'. It displays the phone number '(320) 299-0578'. To the right, there's a link 'Don't like this one? Search for a different number'. Below the number, there's a list of capabilities: 'This United States phone number has the following capabilities:' followed by icons for 'Voice', 'SMS', and 'MMS'. At the bottom of the modal are two buttons: 'Cancel' and a red 'Choose this Number' button.

Готово!
Вы получили номер, с которого будут уходить SMS-сообщения:

Congratulations!

X

Your new Phone Number is **+13202990578**

For help building your Twilio application, check out the resources on the getting started page.
Once you've built your application, you can configure this phone number to send and receive calls and messages.

Done

Ограничения бесплатного режима:

- Сообщения можно отправлять только на номера, от которых получено разрешение на рассылку
- Любое сообщение будет начинаться с "Sent from a Twilio trial account"

Теперь проверим, что всё это работает и отправим SMS-сообщение на свой номер.

Для выполнения задания вам понадобится токен и документация по API. Токен сохранён в специальном разделе вашего аккаунта, а документация по API Twilio лежит здесь.

Практика

Создайте и активируйте виртуальное окружение, установите библиотеку для работы с Twilio API (в документации сервиса Twilio есть инструкция) и напишите код, который генерирует запрос к API Twilio и отправит SMS на ваш номер телефона.

Потестируйте, что будет при отправке SMS на другой телефон [спойлер: никто не пострадает от незапрошенных SMS]

Не забудьте спрятать секретную информацию в переменные окружения: это account_sid, auth_token и номера телефонов. В случае затруднений обращайтесь к документации, там всё есть.

- Импортируйте класс `Client` из пакета `twilio.rest`
- Создайте экземпляр этого класса и передайте в него **account_sid** и **auth_token** в качестве аргументов (их значения можно взять в консоли Twillo).
- Вызовите у созданного экземпляра метод `messages.create` и передайте в него текст для отправки.

Результат: на ваш телефонный номер пришло SMS-сообщение с переданным текстом.

53_Django REST Framework

REST API можно реализовать на чистом Python или написать его на Django. А разработку API на Django можно ускорить и упростить, подключив библиотеку *Django REST Framework*.

Django REST Framework (DRF) предоставляет весь необходимый набор инструментов для создания REST-сервисов на основе Django. По сути, **DRF** — это набор предустановленных классов, сходных с *Generic Views*, но они работают с API. Также DRF включает инструменты для сериализации, аутентификации и для решения других штатных задач, возникающих при создании REST API.

На основе **DRF** можно минимальным количеством кода преобразовать любое Django-приложение в REST API.

Этим вы и займитесь — создадите для **Mimdb** API-интерфейс, с которым могут работать мобильное приложение или чат-бот.

В качестве финального проекта этого курса вы с нуля построите REST API для проекта **Mimdb** — сервиса, где пользователи смогут публиковать отзывы о фильмах и ставить фильмам оценки. Проект не будет предоставлять возможность смотреть фильмы, но будет содержать название, короткое описание и баннер фильма. REST-ресурс, описывающий такую сущность, мы будем для простоты называть «фильм».

DRF очень хорошо документирован, и, несмотря на наличие других библиотек, стал стандартом в разработке REST-сервисов на Django.

Откройте и полистайте [документацию по Django REST Framework](#). Уже сейчас, в начале изучения библиотеки DRF, обзор документации поможет вам понять, на что способен этот фреймворк. Сохраните эту ссылку, она не раз вам пригодится.

54_ Сериализаторы

Сериализацией называют процесс преобразования данных из одного формата в другой. Например, можно сериализовать экземпляр класса в объект JSON и обратно. Вместо JSON может выступать любой другой формат данных, но именно JSON является стандартом в индустрии и применяется в Django REST Framework по умолчанию.

Работа с запросами к API устроена так: программа-клиент отправляет запрос со списком параметров на определенный URL. API разбирает эти параметры и отдаёт клиенту запрошенную информацию. Разбор запроса и создание ответа происходит во view-функции или view-классе, написанных на Python. И для того, чтобы разобрать запрос и создать ответ — применяют классы-сериализаторы, переводчики с JSON на Python и обратно.

По принципу работы сериализаторы очень похожи на формы (Forms), с которыми вы уже имели дело: они переводят *querysets* (списки объектов) или отдельные объекты модели в формат JSON и могут валидировать данные.

Сериализатор не только преобразует данные в JSON, но и указывает, какие поля включить в выдачу или исключить из неё.

Как и формы, сериализаторы могут быть связаны с моделями, но это не обязательно. Не связанные с моделями сериализаторы могут обрабатывать произвольные данные, например, объекты классов Python.

В коде это будет понять проще, чем в теоретическом описании.

Возьмём для примера класс Post, который описывает сущность публикации. У этого класса есть три поля — автор (для упрощения пусть это будет строка), текст и дата публикации. Чтобы в ответ на запрос к API вернуть структурированную информацию о записи, нужно сериализовать объект этого класса в формат JSON.

Создадим класс **Post** и опишем класс-сериализатор **PostSerializer**. А затем создадим экземпляр класса Post и передадим его в конструктор сериализатора. Теперь сериализованные данные хранятся в виде словаря в `serializer.data`

```

from datetime import datetime
from rest_framework import serializers

class Post():
    def __init__(self, author, text, pub_date=None):
        self.author = author
        self.text = text
        self.pub_date = pub_date or datetime.now()

# создаём сериализатор, наследник предустановленного класса Serializer
class PostSerializer(serializers.Serializer):
    author = serializers.CharField(max_length=200)
    text = serializers.CharField(max_length=5000)
    pub_date = serializers.DateTimeField()

# создаём объект класса Post
post = Post(author="Лев Толстой", text="17 Марта. Написаль около листа Юности
хорошо, но могъ бы написать больше и лучше. – И легъ поздно.")

# передаём объект класса Post в сериализатор, чтобы превратить данные объекта
# в JSON
serializer = PostSerializer(post)

# смотрим, что получилось
print(serializer.data)
# будет напечатано: {"author": "Лев Толстой", "text": "Текст", "pub_date":
"2020-03-23T18:02:33.123543Z"}
# это же JSON!

```

Код сериализаторов обычно сохраняют в файле *serializers.py*.

55_View-функции API

Запрос к API проходит такой путь:

urls.py — здесь настраивается маршрутизация API, описываются эндпоинты для ресурсов; **Эндпоинтами** в терминах REST API принято называть URL-адрес, идентифицирующий ресурс.

serializers.py — здесь данные преобразуются в JSON и обратно;

views.py — здесь идёт обработка параметров запроса, получение или запись данных, создание ответа.

Как и всегда в Django, в работе с API применяются Views: функции или классы. Классы могут быть низкого уровня (их пишут вручную) или «высокоуровневые»: в них заготовлены инструменты для решения типовых задач. Основная идея в том, что для решения экзотических задач

применяют низкоуровневые классы, а если задача типичная (скажем, CRUD) — высокоуровневые.

Есть ещё так называемые **мета-функции**, или **вьюсеты**. Это очень мощный инструмент, ему будет посвящена отдельная тема нашего курса.

Token Authentication

Один из принципов построения REST — отсутствие состояния (*stateless*). Это значит, что каждый запрос к серверу должен содержать информацию о том, кто совершает этот запрос (информацию о пользователе).

Сопоставление этих данных с учетными данными на сервере называется аутентификацией.

Рассмотрим самый простой способ аутентификации — по токену.

Для получения токена клиент посылает серверу запрос, передав логин и пароль. Если в базе данных существует такой пользователь и пароль совпадает, то в ответ клиент получает токен. При каждом последующем запросе к API проводится проверка: есть ли переданный с запросом токен в базе и какому пользователю он соответствует.

Для реализации этого механизма нужно выполнить несколько действий.

- Добавить приложение 'rest_framework.authtoken' в **INSTALLED_APPS**
- В `settings.py` объявить новый способ аутентификации `TokenAuthentication` и, одновременно, запретить доступ всем неаутентифицированным пользователям, установив значение `IsAuthenticated` для ключа `DEFAULT_PERMISSION_CLASSES`:

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ],

    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
    ]
}
```

Ограничение доступа настраивается с помощью **пермишенов** (англ. "permissions", разрешения). Мы поговорим о них подробно в следующем спринте, а пока просто подключите их к проекту.

- Выполнить миграции, чтобы в базе данных создались поля для работы и хранения токена: `python manage.py migrate`

- Добавить маршрут для получения токена:

```
from rest_framework.authtoken import views

urlpatterns += [
    path('api-token-auth/', views.obtain_auth_token)
]
```

В этом уроке вы будете работать в тренажере, поэтому миграции мы выполнили за вас.

Теперь можно отправить POST-запрос к адресу `api-token-auth/`, передать в теле запроса поля `username` и `password` — и в ответ придёт заветный токен:

```
{ "token": "d31754b63d74f15b48aa91ac387f5115d0ba83d6" }
```

Этот токен надо будет передавать в заголовке каждого запроса, в поле **Authorization**: `Authorization: Token d31754b63d74f15b48aa91ac387f5115d0ba83d6`

При успешной аутентификации `TokenAuthentication` вернёт такие данные:

- `request.user` будет экземпляром пользователя.
- `request.auth` будет токеном
(экземпляром `rest_framework.authtoken.models.Token`)

В ответ на неаутентифицированные запросы клиент получит отказ: ему в ответ придёт статус-код HTTP 401 Unauthorized:

```
{ "detail": "Authentication credentials were not provided." }
```

View-функции

Чтобы создать view-функцию для API, нужно декорировать обычную view-функцию следующим образом: `@api_view(<разрешенные HTTP-методы>)`. Такая функция должна возвращать объект `Response(<отправляемые данные>)`.

Рассмотрим пример. View-функция `hello` при GET-запросе возвращает ответ «Привет, мир!» в формате JSON. При POST-запросе она вернёт строку со включением полученных данных, если они были. При запросах другого типа, например, PUT, вернётся сообщение «Метод не разрешен» и статус-код ошибки "405 Method Not Allowed".

```

from rest_framework.decorators import api_view
from rest_framework.response import Response

@api_view(['GET', 'POST'])
def hello(request):
    if request.method == 'POST':
        return Response({'message': f'Привет {request.data}'})
    return Response({'message': 'Привет, мир!'})

```

Обычно в API создание нового объекта и запрос всех объектов реализуется в одной функции, а получение/изменение/удаление объекта — в другой. То есть весь функционал CRUD может быть реализован двумя view-функциями API.

Для удобства работы и отладки у DRF есть специальный браузерный API. Он позволяет совершать все виды запросов прямо из браузера, красиво и структурированно выводит объекты, поддерживает авторизацию. Используйте его для проверки работоспособности ваших эндпоинтов.

The screenshot shows the Django REST framework's browsable API interface. At the top, there is a navigation bar with the text "Django REST framework" and "admin". Below it, a header for "Api Posts" is displayed, along with "OPTIONS" and "GET" buttons. A "GET /api/v1/posts/" button is also present. The main content area shows a successful "HTTP 200 OK" response with the following JSON data:

```

HTTP 200 OK
Allow: POST, OPTIONS, GET
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "text": "fadfsdf",
        "author": 1,
        "image": null,
        "pub_date": "2020-03-23T19:10:46.556950Z"
    },
    {
        "id": 2,
        "text": "fsdfsfs",
        "author": 1,
        "image": null,
        "pub_date": "2020-03-23T19:10:50.967999Z"
    }
]

```

Below this, there is a "Media type:" dropdown set to "application/json" and a "Content:" text area. A "POST" button is located at the bottom right of the form.

Существуют также другие мощные инструменты для удобной работы и тестирования API. Например, консольный `HTTPie` и графический `Postman`. Потестируйте их, эти сервисы помогут вам в разработке.

Пришло время самостоятельно реализовать CRUD на Django REST Framework с помощью view-функций API.

Работайте с сериализатором так же, как с уже знакомыми вам **формами**. Вспомните методы `is_valid()`, `save()`, и `delete()` — и у вас всё получится! Прежде чем вы отправитесь в бой, отметим ещё несколько важных моментов:

- View-функция обрабатывает только те запросы, которые вы указали в декораторе `@api_view(<разрешенные HTTP-методы>)`
- Данные приходят в объекте `request.data` и передаются в конструктор сериализатора через именованный параметр `data`.
- Необходимо вернуть клиенту правильный статус-код. Список кодов можно уточнить [здесь](#).

56_View-классы API

Знакомство со view-классами **Django REST Framework** начнём с низкоуровневого `APIView` из модуля `rest_framework.views`.

Как и его класс-родитель `View`, класс `APIView` при получении запроса вызывает пустой метод. При получении GET-запроса будет вызван метод `get()`, при получении POST-запроса — метод `post()`, но эти методы не выполняют никаких действий, их функциональность нужно описывать самостоятельно. В целом этот класс работает так же, как и view-функции:

```
class APIPost(APIView):
    # переопределим родительский метод get(), всё равно он был пустой и
    # ничего не умел
    def get(self, request):
        posts = Post.objects.all()
        serializer = PostSerializer(posts, many=True)
        return Response(serializer.data)
    # переопределим родительский метод post(), с ним та же история, что и с
    # get()
    def post(self, request):
        serializer = PostSerializer(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)
```

Как вы заметили, код ничем не отличается от того, что был во view-функции, только он описан в объектно-ориентированном стиле.

Низкоуровневые view-классы нужны для решения специфических задач, описав с нуля всю логику (в низкоуровневых классах логика по умолчанию не предустановлена).

При типовых действиях (вывод списка объектов или запрос объекта по *id*) удобнее использовать высокоуровневые view-классы, «дженерики» (*Generic Views*): в них уже реализованы все механизмы, необходимые для решения задачи.

Некоторые из них выполняют строго определённую задачу, а некоторые более универсальны и могут «переключаться» на разные задачи в зависимости от HTTP-метода, которым был отправлен запрос.

В дженериках задают два поля: `queryset` (набор записей, который будет обрабатываться) и `serializer_class` (сериалайзер, который будет преобразовывать объекты в формат JSON). В DRF все *Generic Views* объединены в модуле `rest_framework.generics`. Полный список можно посмотреть [здесь, в документации по django rest framework](#).

Рассмотрим для примера класс `ListCreateAPIView`: он отображает всю коллекцию (все посты) или может создать новую запись в модели (новый пост в модели `Post`, например).

А для того, чтобы просматривать, обновлять или удалять записи по одной — применим дженерик-класс `RetrieveUpdateDestroyAPIView`.

Опишем их в коде:

```
from rest_framework import generics  
  
from .models import Post  
from .serializers import PostSerializers  
  
class PostList(generics.ListCreateAPIView):  
    queryset = Post.objects.all()  
    serializer_class = PostSerializer  
  
class PostDetail(generics.RetrieveUpdateDestroyAPIView):  
    queryset = Post.objects.all()  
    serializer_class = PostSerializer
```

Работа с DRF отличается от знакомой вам структуры только сериализацией (файл `serializers.py`) и отсутствием работы с шаблонами. Мы сделали API для **Poemnotes** при минимальном количестве изменений в коде!

Удобная шпаргалка по классам DRF: <http://www.cdrf.co/> Добавьте эту страницу в закладки и заглядывайте туда регулярно: с каждым разом этот справочник будет становиться всё понятнее и полезнее.

В примере кода мы унаследовались от комбинированных view-классов: их удобнее применить к решению нашей задачи. Но иногда они бывают избыточны и требуется view-класс, выполняющий только одно действие. Например, при создании API «только для чтения» (как знакомый вам StarWarsAPI) лучше подключить `ListAPIView`, который выполняет ровно одно действие: выводит список объектов в ответ на метод GET.

Вот остальные view-классы DRF, которые выполняют только одно действие:

- `RetrieveAPIView` — выводит один объект (метод GET)
- `CreateAPIView` — создает новый объект (метод POST)
- `UpdateAPIView` — редактирует объект (методы PUT и PATCH)
- `DestroyAPIView` — удаляет объект (метод DELETE)

Работа с ними аналогична работе с комбинированными view-классами модуля `rest_framework.generics`.

57 CRUD для Poemnotes

Viewsets

Viewset — это view-класс, реализующий все операции CRUD: вывод объекта или списка объектов, добавление, правка и удаление объекта. Viewset заменяет два обычных view-класса.

Во viewsets встроена обработка разных типов запросов, работа с сериализаторами и моделями, возврат ошибок. Не нужно ничего придумывать и писать повторяющийся код, как вы это делали раньше. Чтобы изменить для каких-то методов поведение по умолчанию, достаточно в классе-наследнике переопределить нужный метод и внести в него необходимые изменения.

Кроме того, вьюсеты предоставляют инструмент для генерации эндпоинтов, так называемые **роутеры** (*routers*).

Всё, что нужно сделать — импортировать во `views.py` класс `ModelViewSet` из модуля `viewsets` и создать класс-наследник. Внутри класса нужно описать два обязательных поля — `queryset` и `serializer_class`:

```
class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()
    serializer_class = UserSerializer
```

Чтобы создать роутер, который генерирует эндпоинты, нужно в файле `urls.py` импортировать класс `DefaultRouter` из модуля `routers` и создать объект этого класса. Затем для регистрации каждого эндпоинта нужно вызывать у объекта роутера метод `register()`, который принимает в качестве аргументов для параметра: шаблон URL и класс вьюсета. И включить роутер в список `urlpatterns`:

```
from rest_framework.routers import DefaultRouter

router = DefaultRouter()
router.register('api/v1/users', UserViewSet)

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

В случае затруднений обращайтесь к документации: <https://www.django-rest-framework.org/api-guide/viewsets/>. Там же можно посмотреть примеры применения.

Есть и неофициальный перевод на русский язык (но лучше привыкать читать документацию на английском): <https://github.com/ilyachch/django-rest-framework-rusdoc/blob/master/api-navigation/viewsets.md>

Про CORS

При деплое вашего проекта на веб-сервер вы можете столкнуться с ограничениями на отправку запросов с других доменов.

По умолчанию отправка запросов ограничивается пределами одного домена: программа-клиент, отправляющая запрос к серверу, должна находиться в том же домене, что и веб-сервер, к которому направлен запрос. Это называется «политика единого источника».

При взаимодействии программы-клиента с API, находящимся в другом домене (например, клиент размещён на *mysite.ru*, а API – на *yoursite.ru*) – потенциально возможны проблемы безопасности.

Вы уже сталкивались с этой проблемой при отправке форм. Для защиты от поддельных запросов вы создавали и отправляли вместе с формой CSRF-токен, по которому сервер определял, что запрос не подделан.

По умолчанию кросс-доменные запросы запрещены в дефолтных настройках веб-сервера: это самое простое решение. Но тут возникает другая проблема: к открытому API надо разрешить запросы с любого домена, иначе какой же он «открытый».

Чтобы к нашему API могли обращаться любые приложения – нужно разрешить фреймворку использовать **CORS**.

Cross-Origin Resource Sharing (англ. «совместное использование ресурсов между разными источниками») – это разрешение на обработку запросов с другого домена.

Для настройки **CORS** установите пакет *django-cors-headers* в виртуальном окружении:

```
pip install django-cors-headers
```

Подключите его в *settings.py* как приложение:

```
INSTALLED_APPS = [
    ...
    'rest_framework',
    'corsheaders',
]
```

И в списке *MIDDLEWARE* зарегистрируйте обработчик (обязательно перед *CommonMiddleware*)

```
MIDDLEWARE = [
    ...
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
    ...
]
```

Осталось добавить в конфиг две настройки: `CORS_ORIGIN_ALLOW_ALL` и `CORS_URLS_REGEX`:

- `CORS_ORIGIN_ALLOW_ALL`: значение `True`, установленное для этого ключа, разрешит обрабатывать запросы, приходящие с любого хоста, игнорируя политику *Some Origin*. Если установить `False` или просто удалить этот ключ из конфига — будут разрешены только запросы с текущего хоста.
- `CORS_URLS_REGEX`: значением этого ключа должно быть регулярное выражение, которое определяет URL'ы, к которым можно обращаться с других доменов.

Можно разрешить кросс-доменные запросы к любым адресам домена, но это повысит уязвимость проекта. Потому включим CORS только для путей с префиксом `/api`. В ключе `CORS_URLS_REGEX` задаём регулярное выражение, описывающее такие пути:

```
CORS_ORIGIN_ALLOW_ALL = True  
CORS_URLS_REGEX = r'^/api/.*$'
```

Для ключа `CORS_URLS_REGEX` можно указать и несколько значений через запятую, каждое должно быть в кавычках.

Префикс `r` перед строкой определяет **r-строку**. Такую строку система будет читать как простую последовательность символов, игнорируя escape-последовательности (*escape sequence*) — комбинации обратного слеша и символа. Например, escape-последовательность `\n` интерпретируется как перенос строки, но в r-строке это будут просто два текстовых символа безо всякого скрытого смысла.

Настройка доступа к API завершена. Теперь любой клиент с любого домена может отправлять запросы к вашему API.

Важно: если вы делаете закрытый API, то в `CORS_ORIGIN_WHITELIST` укажите домен, с которого разрешены запросы (например, `localhost:3000`) и удалите ключ `CORS_ORIGIN_ALLOW_ALL` (тогда он примет значение по умолчанию: `False`).

Полное руководство по работе с `django-cors-headers` находится здесь: <https://github.com/adamchainz/django-cors-headers>

58_Авторизация (проверка прав)

При действующем уровне доступа к вашему API любой аутентифицированный пользователь может удалять и редактировать не только свои, но и чужие записи. Это неправильно.

В курсе Django вы применяли декоратор `@login_required` для ограничения доступа к определённым страницам проекта. В прошлом спринте вы узнали, как ограничить доступ к API для неаутентифицированных пользователей. Но и этого недостаточно.

Безопасность — важная часть любого сервиса, и если вы не настроите механизмы авторизации, любой пользователь получит полный доступ ко всем функциям API.

Самое время разобраться со встроенными разрешениями и настроить права доступа так, чтобы редактировать и удалять записи могли только их авторы.

Разрешения (Permissions)

При запросе к API одновременно с аутентификацией система определяет, достаточно ли прав у пользователя на выполнение запрошенных операций. Эта проверка выполняется в самом начале обработки запроса.

Настроить права доступа (или «пермишены», в таком виде этот термин тоже будет вам встречаться) можно на уровне всего проекта, на уровне отдельного приложения или даже на уровне отдельных классов.

Начнём с уровня проекта: будет надёжнее установить глобальные ограничения, а потом ослабить их там, где это необходимо.

Для установки ограничений на уровне проекта в словаре настроек REST_FRAMEWORK задайте параметр `DEFAULT_PERMISSION_CLASSES`

`settings.py`

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

На уровне проекта можно установить один из четырёх вариантов доступа:

- `AllowAny` — всё разрешено, любой пользователь (и даже аноним) может выполнить любой запрос.
- `IsAuthenticated` — только аутентифицированные (зарегистрированные) пользователи имеют доступ к API и могут выполнить любой запрос.
- `IsAuthenticatedOrReadOnly` — неаутентифицированные пользователи могут совершать запросы на чтение, но не могут ничего создавать, удалять или изменять.
- `IsAdminUser` — только администраторы/суперпользователи могут делать запросы.

Чтобы настроить разрешения на уровне view-класса, импортируйте модуль `permissions` из пакета `rest_framework` и добавьте поле `permission_classes` в тело класса:

```
class PostViewSet(viewsets.ModelViewSet):  
    queryset = Post.objects.all()  
    serializer_class = PostSerializer  
    permission_classes = (permissions.IsAuthenticatedOrReadOnly,)
```

Создание собственных разрешений

Иногда возникает необходимость в написании собственных разрешений. Например, вы хотите, чтобы редактировать или удалять пост мог только администратор или автор, а всем прочим пользователям пост должен быть доступен только для чтения.

В Django REST Framework есть базовый класс `BasePermission`, от него наследуются все классы разрешений. Его методы принимают на вход:

- объект запроса `request`: содержит все данные запроса
- объект `view`: объект (класс или функция), к этому объекту можно обратиться, чтобы проверить какие-то поля или методы
- объект `obj`: объект, права на доступ к которому проверяются

Этот класс применяют для ограничения доступа на уровне объектов. Доступ будет разрешён, если метод `BasePermission` вернёт `True`.

```
class BasePermission(metaclass=BasePermissionMetaclass):
    """
    A base class from which all permission classes should inherit.
    """

    def has_permission(self, request, view):
        """
        Return `True` if permission is granted, `False` otherwise.
        """
        return True

    def has_object_permission(self, request, view, obj):
        """
        Return `True` if permission is granted, `False` otherwise.
        """
        return True
```

Чтобы создать собственное разрешение — опишите свой класс, расширяющий **BasePermission** и переопределите один из его методов.

Пример класса, который позволяет любые действия администратору, а больше никому:

```
class IsSuperuserPermission(permissions.BasePermission):
    def has_permission(self, request, view):
        return request.user.is_superuser
```

Чтобы применить ограничения к view-классу, импортируйте `IsSuperuserPermission` во `views.py` и добавьте поле `permission_classes` во view-функцию:

```
from .permissions import IsSuperuserPermission
```

```
class PostViewSet(viewsets.ModelViewSet):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
    permission_classes = (IsSuperuserPermission,)
```

Встроенный в Django метод `.is_superuser` вернёт `True`, если пользователь из `request.user` — администратор.

Если метод вернёт `False`, запрос не будет обработан и вернётся такое сообщение:

```
{
    "detail": "You do not have permission to perform this action."
}
```

59_Throttling (ограничение количества запросов)

Как и в случае с *permissions*, **тrottлинг** определяет, разрешить ли запрос к API. Отличие в том, что он устанавливает ограничение на лимит запросов и определяет разрешённую частоту обращений к API. Это нужно для того, чтобы контролировать нагрузку на сервер и отсекать клиентов, которые слишком часто отправляют запросы.

Можно установить лимит на количество запросов в секунду, минуту, час или день. Также можно определить несколько лимитов для одного клиента, например 1000 в день и 100 запросов в час. Можно создать ограничения на определенные действия, например, на загрузку фотографий на сервер.

При превышении лимита возвращается статус-код **429 "Too Many Requests"**. Если пользователь не аутентифицирован, то для идентификации пользователя запоминается ip-адрес, с которого приходят запросы.

Лимит запросов можно установить для всего проекта, можно — для конкретных view-функций, а можно скомбинировать оба эти подхода. Распространённая практика — задать глобальные настройки для всего проекта, а затем уточнить их на уровне классов или функций.

Лимиты на уровне проекта

Чтобы глобально ограничить неаутентифицированных пользователей тысячей запросов в сутки, а аутентифицированным разрешить десять тысяч — в файле *settings.py* добавьте в словарь REST_FRAMEWORK параметры DEFAULT_THROTTLE_CLASSES и DEFAULT_THROTTLE_RATES:

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

Пользователь, который не прошел процедуру аутентификации, в Django называется анонимным, *anonymous*, сокращенно — *Anon*.

В параметре `DEFAULT_THROTTLE_CLASSES` мы регистрируем классы пользователей (`User` и `Anon`), а в `DEFAULT_THROTTLE_RATES` устанавливаем для них ограничения в формате количество_запросов/период_времени.

Количество запросов — это целое число, период времени указывается как `second`, `minute`, `hour` или `day`.

Имя `user` предустановлено в классе `UserRateThrottle`, а `anon` — в `AnonRateThrottle`

Лимиты на уровне view

Для того, чтобы изменить лимиты на уровне view-функций, есть декоратор `@throttle_classes()`: он принимает на вход классы из `rest_framework.throttling`.

Но если для глобальных настроек мы зарегистрировали эти классы в `settings.py`, то для лимитирования числа запросов на уровне view эти классы указывают там, где необходимо их подключить.

settings.py

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': [
        # не будем подключать классы глобально
        # подключим их только в тех view-классах, где надо установить
        лимиты
    ],
    'DEFAULT_THROTTLE_RATES': {
        # но сами лимиты установим, и они будут доступны из всего кода
        проекта
        'user': '10000/day', # лимит для UserRateThrottle
        'anon': '1000/day', # лимит для AnonRateThrottle
    }
}
```

views.py

```
@api_view(['GET'])
@throttle_classes([UserRateThrottle])
# здесь подключили класс UserRateThrottle
# и для этого view-класса сработает лимит "10000/day" для залогиненных
пользователей,
# объявленный в settings.py
def example_view(request):
    text = {
        'hello': 'world'
```

```
    }
    return Response(text)
```

В случае с view-классами в тело класса добавляют поле `throttle_classes`:

```
class ExampleView(APIView):
    # здесь подключили класс UserRateThrottle
    # и для этого view-класса сработает лимит "10000/day" для
    # залогиненных пользователей,
    # объявленный в settings.py
    throttle_classes = [UserRateThrottle]

    def get(self, request):
        text = {
            'hello': 'world'
        }
        return Response(text)
```

Чтобы создать свой лимит запросов (так называемый `scope`, «скоуп») для view-классов, подключите в `settings.py` класс `ScopedRateThrottle` и задайте лимит в `DEFAULT_THROTTLE_RATES`:

```
REST_FRAMEWORK = {
    ...
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.ScopedRateThrottle',
    ],
    'DEFAULT_THROTTLE_RATES': {
        ...
        # этот лимит сработает лишь в том view-классе, в котором он будет
        # указан
        # имена (ключи) для scope указывает разработчик, в меру
        # собственной фантазии
        'low_request': '3/minute',
    }
}
```

Затем добавьте этот скоуп в тело класса:

```
class ExampleView(APIView):
    throttle_scope = 'low_request'
```

60_Пагинация (Pagination)

Представьте, что у вас в базе хранится значительное количество публикаций. Запрос `api/v1/posts/` вернёт сразу 100500 объектов, и у клиента будут проблемы: как разобраться в такой груде информации (да и потребности в таком объёме данных обычно не возникает). Поможет **Pagination**, система деления выдачи на части (с аналогичной системой вы уже встречались, настраивая вывод постов на фронтенд).

Пагинацию можно включить на уровне всего проекта, установив ключи `DEFAULT_PAGINATION_CLASS` и `PAGE_SIZE` в словаре настроек `REST_FRAMEWORK`. Они отвечают за подключение пагинатора и число объектов в выдаче. Обратите внимание: нужно установить оба этих параметра.

Начнем с самого простого класса `PageNumberPagination`:

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

Теперь при GET-запросе `http://localhost:8000/api/v1/posts/` вы получите такой результат:

```
{
    "count": 100500,
    "next": "http://localhost:8000/api/v1/posts/?page=2",
    "previous": null,
    "results": [
        {
            "id": 1,
            "text": "Это мой блог на Poemnotes, скорее подписывайтесь на
мою ленту!",
            "author": "sunny_blogger",
            "pub_date": "2020-04-06T13:45:00.941389Z"
        },
        {
            "id": 2,
            "text": "Это нора мизантропа, не стучитесь в двери.
Подписчиков забаню!",
            "author": "black_misanthrope",
            "pub_date": "2020-04-09T15:07:29.868981Z"
        },
        ...
    ]
}
```

Посмотрите на результат запроса: если раньше список объектов был прямо в теле JSON, то теперь объекты вложены в список `results`.

Добавление пагинации изменило структуру выдачи, и если клиенты уже пользуются нашим API — у них будут проблемы: после наших изменений извлечь данные из выдачи не удастся, придётся переписывать обработчики. В такой ситуации будет правильно выпустить следующую версию API (в нашем случае — `api/v2/`), а для прежних клиентов оставить

доступной первую версию (пусть клиенты сами решат, когда перейти на новую).

При включённой пагинации запрос к API можно делать с дополнительным параметром `page`. 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. Эти поля упрощают работу с выдачей: например, в интерфейсе поиска можно вывести информацию «Найдено {count} элементов» и дать ссылки для загрузки предыдущей и следующей страниц с результатами поиска.

Пагинацию можно установить на уровне отдельного view-класса (*Generics* или *Viewsets*), указав класс пагинатора в поле `pagination_class`:

```
from rest_framework.pagination import PageNumberPagination
```

```
class PostViewSet(viewsets.ModelViewSet):  
    queryset = Post.objects.all()  
    serializer_class = PostSerializer  
    permission_classes = (permissions.IsAuthenticatedOrReadOnly,)  
    pagination_class = PageNumberPagination
```

Есть и более гибкий класс для пагинации: `LimitOffsetPagination`.

В случае 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`).

При подключении класса `LimitOffsetPagination` GET-запрос может выглядеть так: `http://localhost:8000/api/v1/posts/?limit=10&offset=5` Такой запрос вернёт 10 объектов, с шестого по пятнадцатый (или меньше, если в результате запроса менее 15 объектов).

Обычно этих двух классов пагинации хватает для решения стандартных задач. Но иногда происходят коллизии. Например, данные могут меняться очень быстро — какой-то объект добавится, какой-то удалится — и порядок выдачи нарушится.

Для этих случаев есть «разбиение на основе курсора»: CursorPagination. Такое разбиение гарантирует неизменный порядок элементов и клиент не увидит один и тот же объект дважды при просмотре.

Это похоже на то, как работает API домашек: при запросе вы передавали *timestamp* и ограничивали периоды выдачи изменения статуса домашки. В CursorPagination действует примерно тот же механизм: сервер запоминает последний объект выдачи и генерирует уникальный URL для следующей страницы, которая начинается с того места, где закончилась предыдущая. Единственное условие — объекты должны быть отсортированы по какому-либо полю (например, по дате создания).

61_Фильтрация, сортировка и поиск

Ваш API стал более безопасен, но пока что он не особо удобен в работе: на любой GET-запрос он возвращает либо отдельный объект, либо все объекты ресурса. Вы научились разбивать выдачу на порции, но хочется добавить гибкости к запросам: фильтровать выдачу по каким-то признакам, сортировать, искать объекты по ключевым словам.

Фильтрация

Мы можем разрешить клиенту фильтровать ресурсы по определённому полю. Для фильтрации списка аккаунтов по *username* дадим пользователям возможность делать GET-запросы такого вида: http://localhost:8000/api/v1/users/?username=some_user_name.

Код, который обработает этот запрос, может выглядеть так:

urls.py

```
path('api/v1/users/<str:username>', UserList.as_view())
```

views.py

```
from rest_framework.views import APIView
from rest_framework.response import Response
from .models import User
from .serializers import UserSerializer

class UserList(APIView):
    def get(self, request, username):
```

```

# через ORM отфильтровать объекты модели User
# по значению параметра username, полученного в запросе
users = User.objects.filter(username=username)
# передать в сериализер результаты фильтрации
serializer = UserSerializer(users, many=True)
# вернуть результат сериализации
return Response(serializer.data)

```

В Generic-классах и Viewsets нужно переопределить их встроенный метод `get_queryset()`, а параметр запроса получить из свойства `self.request.query_params`. Всё остальное работает так же, как и во view-классах:

views.py

```

from .models import User
from .serializers import UserSerializer
from rest_framework import generics

class UserList(generics.ListAPIView):
    serializer_class = UserSerializer

    def get_queryset(self):

        queryset = User.objects.all()
        # добить параметр username из GET-запроса
        username = self.request.query_params.get('username', None)
        if username is not None:
            # через ORM отфильтровать объекты модели User
            # по значению параметра username, полученнго в запросе
            queryset = queryset.filter(username=username)
        return queryset

```

Django REST Framework предоставляет фильтрующие бэкенды для упрощения работы с фильтрацией и поиском. **Backend** («бэкенд») — это механизм, который можно подключить к проекту, чтобы получить дополнительную функциональность. В курсе по Django отправку e-mail вы настраивали именно через специализированный бэкенд.

Для подключения бэкенда фильтрации на уровне всего проекта в `settings.py` нужно добавить ключ `DEFAULT_FILTER_BACKENDS` в словарь настроек `REST_FRAMEWORK`:

```

REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS':
    ['django_filters.rest_framework.DjangoFilterBackend']
}

```

Для подключения бэкенда на уровне отдельного Generic-класса можно добавить поле `filter_backends` в тело view-класса.

```
from rest_framework import generics
import django_filters.rest_framework
from .models import Post
from .serializers import PostSerializer

class PostList(generics.ListCreateAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
    filter_backends = [django_filters.rest_framework.DjangoFilterBackend]

    def perform_create(self, serializer):
        serializer.save(author=self.request.user)
```

Обратите внимание: для бэкенда фильтрации нужна сторонняя библиотека `django-filter`. Установите её через менеджер пакетов `pip` и зарегистрируйте в списке приложений `INSTALLED_APPS`:

settings.py

```
INSTALLED_APPS = [
    ...
    'rest_framework',
    'django_filters',
    # да, библиотека django-filter действительно регистрируется как
    django_filters
]
```

Фильтрующий бэкенд передаёт параметры GET-запроса в Django ORM, а тот преобразует их в SQL-запросы.

Сделать SQL-запрос через GET? Легко!

Чтобы разрешить фильтрацию «по точному совпадению», во view-класс добавляют свойство `filterset_fields`, и в нём указывают поля модели, по которым можно фильтровать.

```
from rest_framework import generics, permissions
from django_filters.rest_framework import DjangoFilterBackend
from .models import Post
from .serializers import PostSerializer

# filterset_fields – подключаем
class PostList(generics.ListCreateAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
    filter_backends = [DjangoFilterBackend]
    filterset_fields = ['text',]
```

Теперь можно сделать, например, такой GET-запрос: `http://localhost:8000/api/v1/posts/?text=Краткость%20%E2%80%94%20сестра%20таланта`

Если у какого-то объекта модели Post содержимое поля `text` **полностью** совпадает со значением параметра `text` GET-запроса — этот объект будет добавлен в выдачу API.

```
[  
    {  
        "id": 293,  
        "text": "Краткость – сестра таланта",  
        "author": "toga",  
        "image": null,  
        "pub_date": "1889-04-11T10:11:20.056969Z"  
    }  
]
```

Последовательности символов, начинающиеся со знака «процент» % — это *URL encoding* («кодировка символов для URL»): способ закодировать и передать через URL те спецсимволы, которые нельзя включать в URL и в GET-параметры. Например, в *URL encoding* пробел записывается как %20 (обычные пробелы в URL запрещены). Таблицу символов и их кодировок можно посмотреть [здесь](#).

Популярная задача в приложениях — фильтрация объектов по диапазону значений какого-то поля: подбор товаров в выбранном диапазоне цен или поиск записей в блоге за определённый период.

Создайте файл `filters.py` и опишите класс фильтра, расширяющего класс `FilterSet`:

```
from django_filters import rest_framework as filters  
from .models import Numbers  
  
class NumberRangeFilter(filters.FilterSet):  
    min_number = filters.NumericRangeFilter(field_name="number",  
                                             lookup_expr='gte')  
    max_number = filters.NumericRangeFilter(field_name="number",  
                                             lookup_expr='lte')  
  
    class Meta:  
        model = Numbers  
        fields = ['min_number', 'max_number']
```

Документация по фильтрам полезна и доступна, читайте обязательно: <https://django-filter.readthedocs.io/>

Поиск

DRF даёт разработчикам инструменты и для поиска.

Поисковый фильтр `SearchFilter` можно подключить к view-классу, а в поле `search_fields` указать поля модели, по которым разрешён поиск. Поля должны быть текстовыми: `CharField` или `TextField`.

```
from rest_framework import generics, filters
from .models import Post
from .serializers import PostSerializer

class PostList(generics.ListAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
    filter_backends = [filters.SearchFilter]
    search_fields = ['text',]
```

По умолчанию включен поиск по частичным совпадениям без учёта регистра. Например, при запросе с параметром `?text=кол` в выдачу попадёт пост, в котором есть слово «Колбаса» (даже с большой буквы).

Можно искать по нескольким совпадениям: в запросе их надо разделить запятыми, без пробелов: `http://localhost:8000/api/v1/posts?search=ночь,улица,фонарь`

При таком запросе в выдачу попадут только те объекты, где есть одновременно все совпадения. Примеры можно посмотреть здесь, в официальной документации: <https://www.djangoproject.org/api-guide/filtering/>

Сортировка выдачи

Результат выдачи можно сортировать, подключив к классу фильтр `OrderingFilter` и добавив поле `ordering_fields`:

```
class PostList(generics.ListAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
    filter_backends = [filters.OrderingFilter]
    ordering_fields = ['pub_date', 'username']
```

`?ordering=username` — сортировка выдачи по пользователям в алфавитном порядке.

`?ordering=-username` — сортировка выдачи по пользователям в обратном алфавитном порядке.