Инструкция: после клонирования

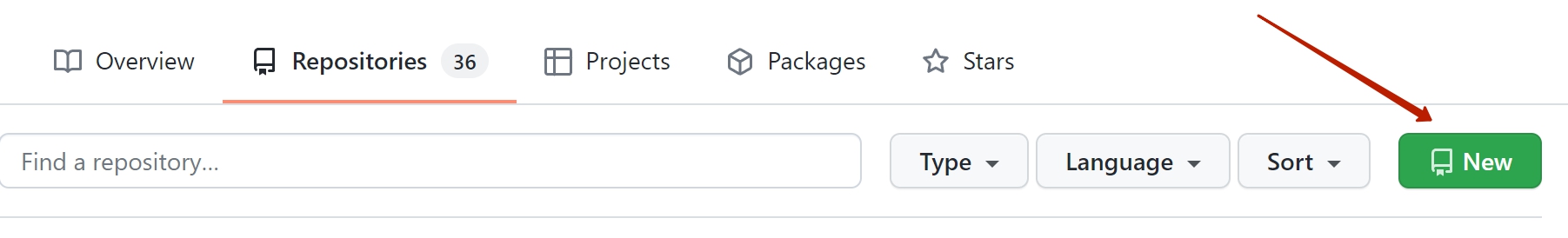
Вступление

Вы клонировали чей-то репозиторий, выполнив команду:  
$ **git remote clone** *<path-to-repo>*

, где *<path-to-repo>* путь до клонируемого репозитория

Теперь вам нужно создать копию этого репозитория на своем github’е

## Инструкция

1. $ **git remote rm origin** - удаляем ссылку на репозиторий с которого клонировали
2. Заходим на [github.com](https://github.com/) и создаем новый(свой) репозиторий  
   
3. Добавляем ссылку на свой репозиторий:  
   
4. Добавляем ssh-ключ, см инструкцию “Добавление SSH-ключа на github”
5. Отправляем проект на свой github:  
   $ **git push -u origin master**
6. Обновляем страницу репозитория в браузере, видим код проекта

Добавление SSH-ключа на github

[Офф-инструкция](https://docs.github.com/en/github/authenticating-to-github/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent)

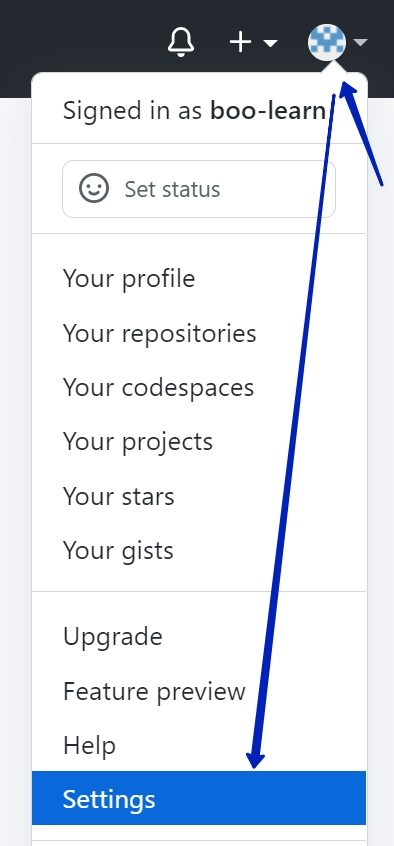
1. Сгенерируйте ssh-ключ: $ ssh-keygen -o
2. Выполните: cat ~/.ssh/id\_rsa.pub  
   В терминале отобразится ключ, скопируйте этот ключ в буфер: ctrl+shift+V
3. Перейдите на: <https://github.com/settings/profile>
4. Найти опцию “SSH and GPG keys” → кнопка “New SSH key”
5. Добавляете свой SSH-ключ(он у вас в буфере) и даете название(произвольное)

Добавление SSH-ключа на github(Linux)

[Официальная инструкция](https://docs.github.com/en/github/authenticating-to-github/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent) на github

1. Сгенерируйте ssh-ключ: $ ssh-keygen -o
   * Enter file in which to save the key(<тут путь к ключу>): нажимаем Enter
   * Enter passphrase(хочет кодовую фразу или пусто): нажимаем Enter
   * Enter same passphrase again: снова Enter

ssh-ключ создан

1. Скопируйте содержимое id\_rsa.pub в буфер обмена  
   Это можно сделать командой в консоли: $ cat ~/.ssh/id\_rsa.pub   
    и копируем в буфер обмена (Ctrl+Shift+C)  
   **Или** можно скопировать содержимое файла в буфер обмена так:  
   $ xclip -sel c < ~/.ssh/id\_rsa.pub  
   Предварительно установив xclip: $ sudo apt install xclip
2. Перейдите в настройки аккаунта:  
     
   **Или** кликнув на ссылку: <https://github.com/settings/profile>   
   (вы должны быть залогинены)
3. Найти опцию “SSH and GPG keys” → кнопка “New SSH key”
4. Добавляете свой SSH-ключ(он у вас в буфере) и даете название(произвольное)

Инструкция: удаленный GIT репозиторий

Примечание

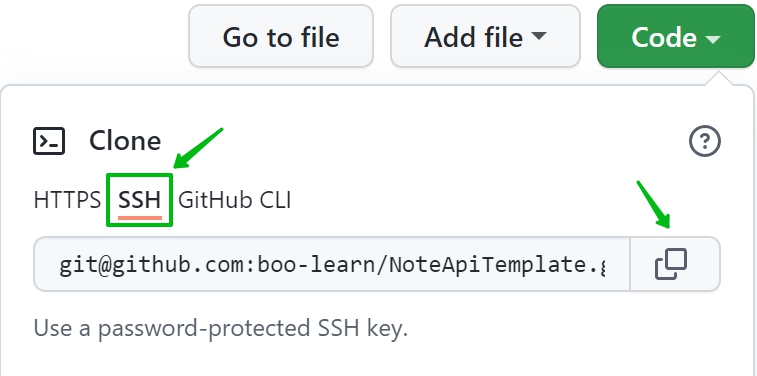
После создания локального репозитория, рекомендуется загрузить свой проект на удаленный репозиторий. Это позволит вам работать с вашим проектом с любого рабочего места.

## Инструкция

1. Регистрируемся на <https://github.com/> и создаем там новый репозиторий (зеленая кнопочка New в правом верхнем углу)
2. Добавляем ssh-ключ на github(см. инструкцию “Добавление SSH-ключа на github”)
3. Выполняем две последние команды(они прописаны в созданном репозитории):

$ **git remote add origin git@github.com:*<you-info>*.git**

$ **git push -u origin master**

Важно! Если вы хотите иметь возможность отправлять изменения на удаленный репозиторий, необходимо добавить именно SSH-ссылку:

1. Обновляем страницу репозитория в браузере, видим код проекта

Дополнительные команды для работы с удаленным репозиторием

$ **git remote rm origin** - удалить ссылку origin на удаленный репозиторий

$ **git remote add origin …** - добавить ссылку на удаленный репозиторий

$ **git pull origin master** - забрать изменения с удаленного репозитория

$ **git push origin master** - отправить изменения на удаленный репозиторий

Инструкция: локальный GIT репозиторий

Примечание

Вся работа производится **в консоли**. Выполняя команды вы **всегда** должны находиться в папке с проектом.

## Инструкция

1. $ **git init** - инициализация(создание) репозитория
2. Создаем .**gitignore** - прописывая в него файлы и папки, которые не должны попасть в репозиторий.
3. Минимальные настройки git’а:   
   $ **git config --global user.name "username"**

$ **git config --global user.email "**[**you@mail.ru**](mailto:you@mail.ru)**"**

1. $ **git add .** - добавляем **все** файлы проекта(кроме тех, что в .gitignore). Подготавливаем файлы к коммиту(сохранению)
2. $ **git commit -m “комментарий к коммиту”** - делаем коммит(сохраняем текущее состояние файлов в репозитории)

Каждый раз, когда вам нужно сохранить изменения в локальном репозитории выполните:

$ **git add .**

$ **git commit -m “комментарий к коммиту”**

Инструкция: Создание venv

## **Создаем виртуальное окружение**

Создаем директорию для нового проекта:

$ mkdir ProjectName  
и переходим в неё:

$ cd ProjectName

Создайте новое окружение:

$ python3 -m venv venv\_name

где **venv\_name** - название окружения

Если получаем ошибку:

|  |
| --- |
| The virtual environment was not created successfully because ensurepip is not  available. On Debian/Ubuntu systems, you need to install the python3-venv  package using the following command.  apt install python3.8-venv  … |

В тексте ошибки написана команда, которую нужно выполнить, чтобы установить программу для создания виртуальных окружения. Выполните эту команду, добавив перед ней sudo(чтобы дать права суперпользователя):  
$ sudo apt update

$ sudo apt install python3.8-venv

, после чего повторить создание виртуального окружения.

## **Активируем виртуальное окружение**

$ source venv\_name/bin/activate

где **venv\_name** - название окружения

Ты поймёшь, что virtualenv запущено, когда увидишь префикс (venv\_name) в начале приглашения командной строки.

**Примечание**: для выхода из виртуального окружения venv наберите:

$ deactivate

Инструкция: Создание проекта Django

*Примечание: Текущие инструкции актуальны для ОС “Linux”*

## Step-0 "create dir"

Создаем директорию для нового проекта Django:

mkdir FirstDjango

и переходим в неё:

cd FirstDjango

## Step-1 "create venv"

Создаем виртуальное окружение (см. “Инструкция: Создание venv”)

***Рекомендация***: для первого проекта назовите виртуальное окружение **django\_venv**

## Step-2 "install Django"

Устанавливаем Django в текущее виртуальное окружение:

(django\_venv)$ pip install django

Если все пройдет без ошибок, получишь сообщение вида:

Successfully installed django-<last-version>

## Step-3 "Check Django install"

1. Убедитесь что вы находитесь/переключились в виртуальное окружение.  
   Ты поймёшь, что virtualenv запущено, когда увидишь префикс (**django\_venv**) в начале приглашения командной строки.
2. Убедитесь что Django нужной версии установлен, для этого:
   1. Запустите интерпретатор python:  
      python
   2. **>>> import django  
      >>> django.VERSION**должно отобразиться:(4, 0, 4, 'final', 0)

## Примечание: **Ctrl+D** - для выхода из режима интерпретатора

## Step-4 "Create Django project"

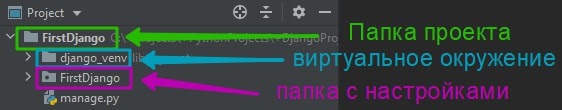
## Перейдите в директорию, которую вы создали в шаге-0(Step-0):

## Для создания нового проекта:

## django-admin startproject FirstDjango .

## где **FirstDjango** название вашего проекта. Точка(.) в конце, говорит “создать проект в текущей директории”

Получаем такую структуру проекта:

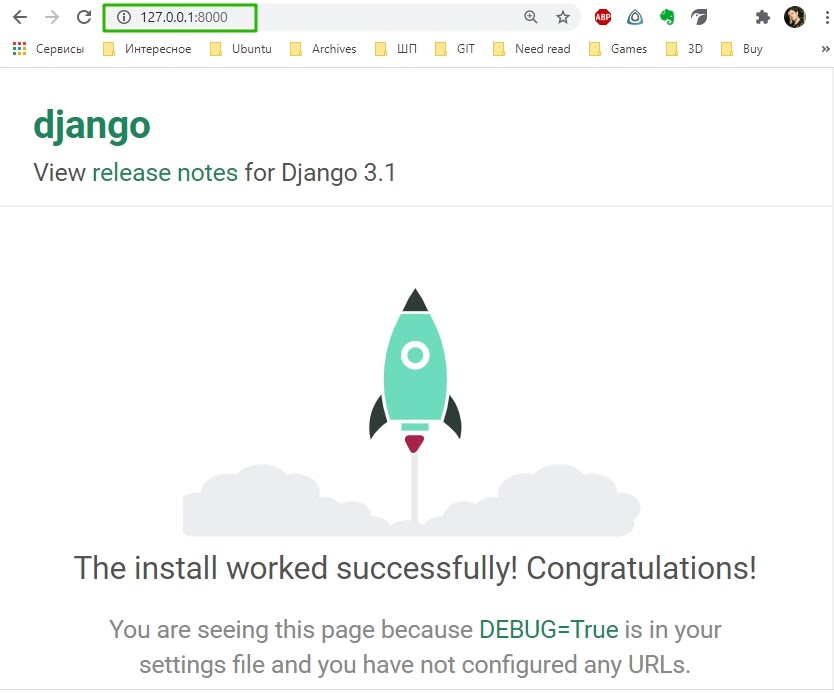


## Step-5 "Run project"

Для запуска проекта:

python manage.py runserver

Перейдите по адресу: <http://127.0.0.1:8000/> в браузере



## Примечание: **Ctrl+C** в консоли- для остановки тестового сервера

## Step-6 "Create App"

Каждый проект должен содержать хотя бы одно приложение

**Проект** – это совокупность приложений и конфигурации сайта.

**Приложение** - часть проекта, реализующая определенный функционал. Приложение может использоваться несколькими проектами.

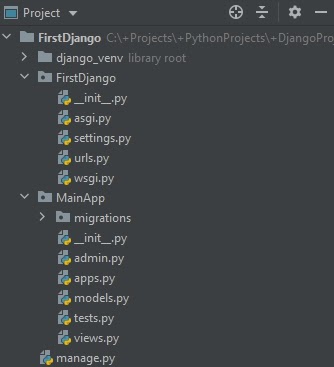
Обычно приложения называются в соответствии с выполняемыми задачами, например: **AuthApp**(отвечает за авторизацию) или **MainApp**(основное приложение).

Для создания нового приложения в текущем проекте:

python manage.py startapp MainApp

Заготовка проекта готова!

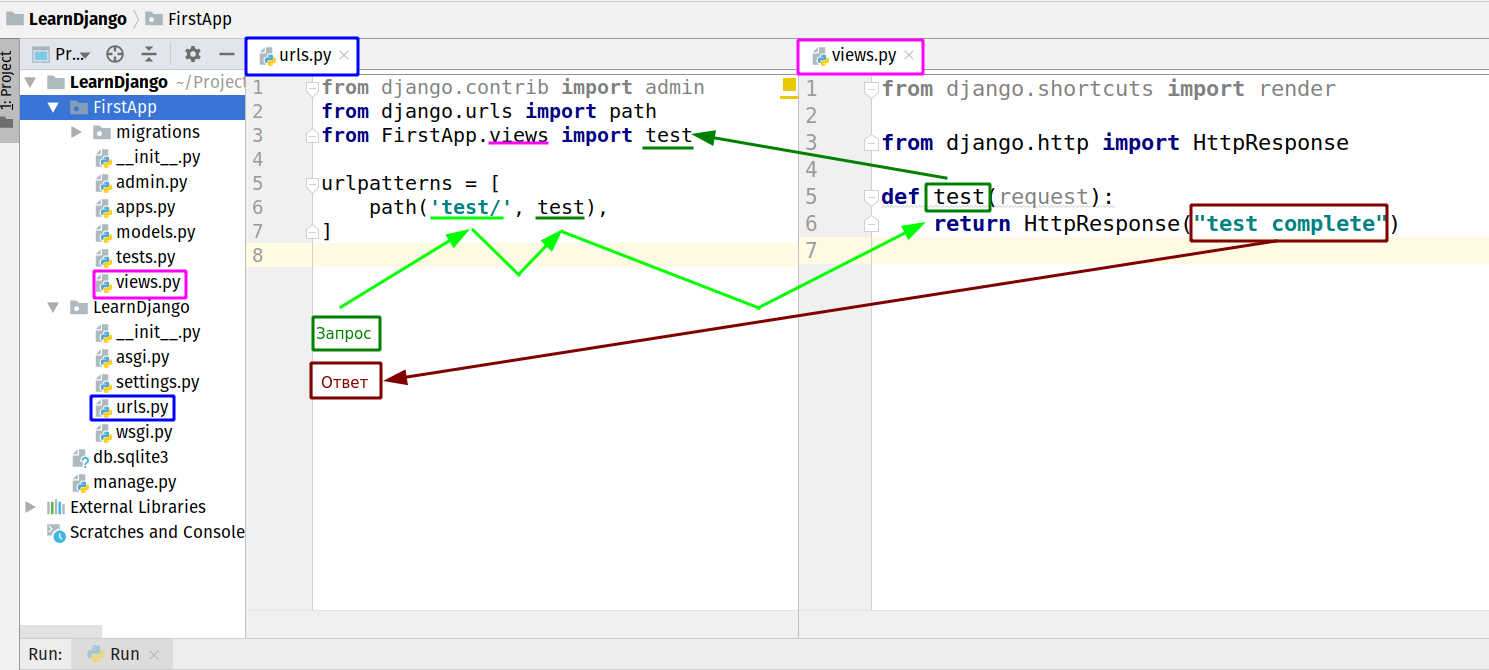
Вот такую структуру файлов вы получите:

  
Можно приступать к работе.

Подробнее можно почитать в [официальной документации](https://django.fun/docs/django/ru/3.0/intro/tutorial01/#writing-your-first-django-app-part-1).

Теория: Обработка запросов

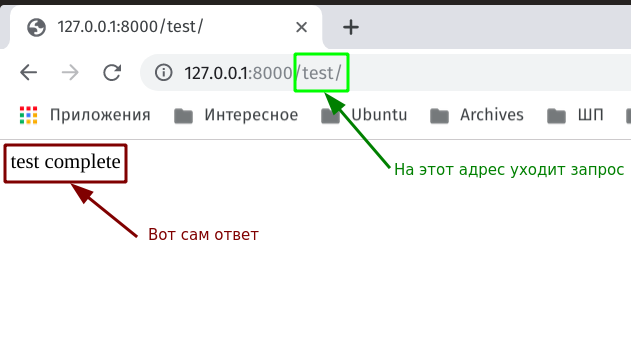
## Демонстрация: простая обработка запроса



Пришедшие запросы обрабатываются файлом **urls.py** и перенаправляются в указанную функцию-обработчик.

**Функция-обработчик** должна вернуть ответ в формате HTTP-протокола. В Django существуют оберточные функции, которые формируют ответ в нужном формате, например функция HttpResponse.

Так результат выглядит в браузере:



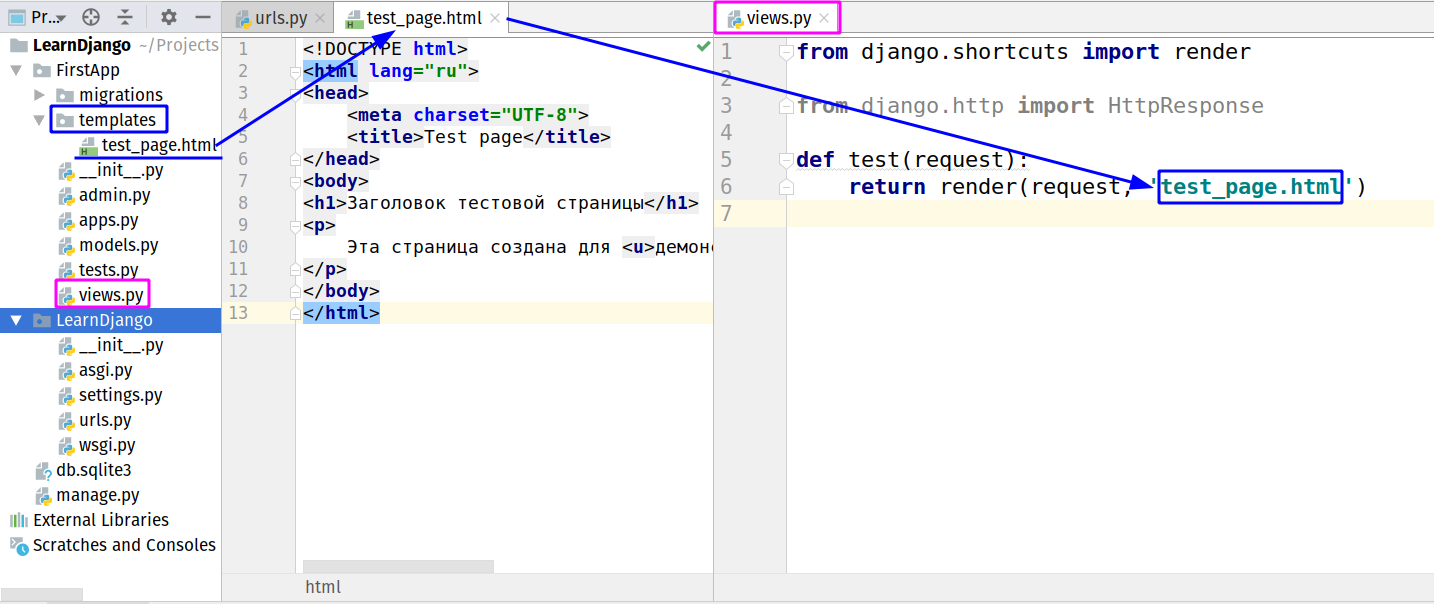
**Напоминание**: Не забудьте запустить django-server!

Важные термины:

* **request** - объект запроса пришедший от клиента
* **response** - объект ответа, который сервер отправит клиенту
* **Функция-обработчик** - python функция, которая получает запрос и возвращает ответ. Обычно находится в файле **views.py**

Теория: Основы работы с шаблонами

## Демонстрация: отправка html-страницы



Для отправки html-страницы на запрос браузера, выполните:

1. Создайте папку **templates** в директории созданного Приложения(не в корне проекта, а именно в папке приложения!)
2. Добавьте ваше приложение(MainApp) в INSTALLED\_APPS  
   **settings.py**

|  |
| --- |
| **INSTALLED\_APPS** = [  **'django.contrib.admin'**,  **...**  **'MainApp'**,  ] |

1. В папке **templates** создайте html-файл (ПКМ на папке new --> HTML file)
2. В html-файле напишите что-нибудь.
3. В функции-обработчике используйте функцию render() (см. пример кода на изображении выше). Функция render() принимает два параметра: первый объект запроса(request), второй строку с названием html-файла.

***Примечание***: Django по умолчанию ищет html файлы в папке **templates**, при желании это можно изменить.

## 

## Демонстрация: вставка контекста(данных) в шаблон



**context** - это просто питоновский словарик.

Все места вставки обозначаются {{ }}, между скобками указываются ключи словарика.

Например, если в качестве контекста передать словарик context = {'a': 5, 'b': -4},

то в шаблоне нужно использовать a = {{ a }} b = {{ b }}, чтобы вывести a = 5 b = -4

Подробнее о шаблонах можно посмотреть в [официальной документации](https://django.fun/docs/django/ru/3.0/ref/templates/language/#templates) или [перевод документации](https://djbook.ru/rel3.0/topics/templates.html#the-django-template-language).

Теория:Шаблон + Контекст = html

Шаблонизатор django.

[Шаблоны и контекст](#_c713xnuw3xem)

[Представления](#_1ph11ss8rzdr)

[Шаблонизатор Django](#_yc53f5hxs8hr)

[Шаблонные фильтры](#_huddx6twu6bm)

# 

# Шаблоны и контекст

Будучи веб фреймворком, Django позволяет динамически генерировать HTML. Самый распространенный подход - использование шаблонов. Шаблоны содержат статический HTML и динамические данные, рендеринг которых описан специальным синтаксисом. Эти данные и называются контекстом.

Проект Django может использовать один или несколько механизмов создания шаблонов (или ни одного, если вы не используете шаблоны). Django предоставляет бэкенд для собственной системы шаблонов, которая называется - язык шаблонов Django (Django template language, DTL),

Django предоставляет стандартный API для загрузки и рендеринга шаблонов, независимо от используемого бэкенда. Загрузка включает в себя поиск шаблона по названию и предварительную обработку, обычно выполняется загрузка шаблона в память. Рендеринг означает передачу данных контекста в шаблон и возвращение строки с результатом.

Язык шаблонов Django – собственная система шаблонов Django. До Django 1.8 – это была единственная альтернатива. Это хорошая система шаблонов, но со своими особенностями.

В качестве практического примера, дополним статических html страницы проекта “About me”, данными ФИО автора. Т.к. у нас пока нет БД, то данные будем хранить в переменных в python-коде.

Допустим на данный момент ваше ФИО на странице располагается как-то так:

|  |
| --- |
| <div class="name">  <h1>  Юрченко Евгений Витальевич  </h1>  </div> |

Добавим данные в **views.py**

|  |
| --- |
| def main(request):  name = 'Евгений'  surname = 'Юрченко'  middle\_name = 'Витальевич'  return render\_to\_response("index.html") |

Чтобы передать эти данные в виде контекста в шаблон:

|  |
| --- |
| def main(request):  name = 'Евгений'  surname = 'Юрченко'  middle\_name = 'Витальевич'  return render\_to\_response("index.html", {'name': name, 'surname': surname, 'middle\_name': middle\_name}) |

Просто передаем их в виде словаря, вторым аргументом функции render\_to\_response

Осталось вставить данные в шаблон:

|  |
| --- |
| <div class="name">  <h1>  {{ surname }} {{ name }} {{ middle\_name }}  </h1>  </div> |

Используем шаблонные теги {{ <ключ словаря контекста> }}, вместо {{ name}} будет подставлено значение словаря контекста под ключем “name”.

Запустим сервер разработки и проверим результат.

Django-шаблонизатор это очень мощный инструмент, подробнее с ним будем знакомиться на протяжении всех уроков. Если вам не терпится, полную информацию можете посмотреть [тут](http://djbook.ru/rel1.8/ref/templates/api.html).

# Представления

Теперь подробнее разберемся с функциями представления, располагающимися в файле views.py.

Каждое представление отвечает за выполнение одной из двух вещей: возвращение объекта HttpResponse, представляющего содержимое запрашиваемой страницы, или вызов исключения такого как Http404.

Как правило, представление извлекает данные в соответствии с заданными параметрами, загружает шаблон и отображает этот шаблон вместе с полученными данными.

И не забывайте, каждая функция-представления первым аргументом получает объект запроса. Этот объект несет всю необходимую информацию о запросе.

Django использует объекты ответа и запроса, чтобы передавать состояние в системе.

Когда запрашивает страница, Django создает объект HttpRequest, который содержит различные данные о запросе. Потом Django определяет и загружает необходимое представление и вызывает его передавая объект HttpRequest первым аргументом. Каждое представление должно вернуть объект HttpResponse.

Т.е. наша функция render\_to\_response() - всего лишь удобная обертка для формирования ответа в соответствии с [протоколом http](https://ru.wikipedia.org/wiki/HTTP). Чтобы браузер корректно обработал ответ, ответ необходимо правильно сформировать, добавив тело, заголовки и прочую техническую информацию, эту рутинную операцию django берет на себя, предоставляя нам удобные инструменты.

Функции-представления содержат бизнес логику приложения(сайта), в то время как шаблоны содержат логику отображения. Если говорить проще: шаблон знает как себя правильно отобразить, а функция-представления знает как получить правильный контекст(данные) для шаблона. В свою очередь модели(речь о которых пойдет в следующих уроках) описывают структуру хранения данных.

# Шаблонизатор Django

Мы уже использовали шаблон переменной {{ }}. Пришло время подробнее познакомиться с шаблонизатором.

Теги выглядят таким образом: {% tag %}. Теги сложнее чем переменные: одни создают текст для вывода, влияют на выполнение используя условия и циклы, другие загружают дополнительную информацию в шаблоны, чтобы использовать ее далее через переменные.

Некоторые теги требуют открывающий и закрывающий теги (например, {% tag %} ... содержимое тега ... {% endtag %}).

Django содержит около двадцати встроенных тегов. Подробно обо всех тегах [тут](http://djbook.ru/rel1.8/ref/templates/builtins.html#ref-templates-builtins-tags). А мы рассмотрим несколько для знакомства на примере нашего проекта.

На данный момент, ваша страница со списком мест где вы работали, выглядит подобным образом:

|  |
| --- |
| <h1>Опыт работы</h1>  <ol>  <li>Skyparkcdn</li>  <li>tranio.ru</li>  <li>НУДО "НАДиП</li>  <li>ООО «Терминал-Сервис»</li>  </ol> |

Понятно, что чтобы добавить или изменить одно из мест работы, вам придется менять код html страницы, а это совсем не то, чего мы хотим. Немного подумав, мы можем сделать так:

views.py

|  |
| --- |
| def works(request):  place\_of\_work = ['Skyparkcdn', 'tranio.ru', 'НУДО "НАДиП', 'ООО «Терминал-Сервис»']  return render\_to\_response("works.html", {'place\_of\_work': place\_of\_work}) |

Используя шаблон переменных, можно сделать так:

works.html

|  |
| --- |
| <h1>Опыт работы</h1>  <ol>  <li>{{ place\_of\_work.0 }}</li>  <li>{{ place\_of\_work.1 }}</li>  <li>{{ place\_of\_work.2 }}</li>  <li>{{ place\_of\_work.3 }}</li>  </ol> |

Уже лучше, но все равно, не то что нам требуется (добавление нового места работы, не добавит запись в списке работ).

Тут отлично подойдут шаблонные теги цикла:

works.html

|  |
| --- |
| <h1>Опыт работы</h1>  <ol>  {% for work in place\_of\_work %}  <li>{{ work }}</li>  {% endfor %}  </ol> |

Очень похоже на python цикл for in.

Все, наша задача решена. Наш шаблон отобразит любое кол-во мест работ из списка, нам больше не нужно изменять код шаблона, если изменятся данные.

Если список с местами работы будет пуст, мы просто ничего не увидим, а, например, мы бы хотели увидеть надпись “пока нигде не работал”, если мест работы нет.

Это легко сделать, используя шаблонный тег {% if %}

works.html

|  |
| --- |
| <h1>Опыт работы</h1>  <ol>  {% if place\_of\_work %}  {% for work in place\_of\_work %}  <li>{{ work }}</li>  {% endfor %}  {% else %}  Пока нигде не работал  {% endif %}  </ol> |

Надеюсь, комментарии излишни.

# Шаблонные фильтры

Допустим, вы хотите быть уверены, что ваши ФИО всегда будут выводиться с большой буквы, даже если вы где-то опечатаетесь (например по невнимательности напишите имя с маленькой буквы). Вы конечно можете добавить код по views, который будет устранять эту досадную проблему. Но это будет неверный подход, т.к. во views у вас появляется код, который влияет только на то, как будет отображена информация. Ведь мы знаем, что за отображение отвечать шаблон. Эту проблему легко можно решить, используя шаблонные фильтры.

Допускаем очепятку во views.py

|  |
| --- |
| def main(request):  name = 'евгений'  surname = 'Юрченко'  middle\_name = 'витальевич'  return render\_to\_response("index.html", {'name': name, 'surname': surname, 'middle\_name': middle\_name}) |

Добавляем фильтр в index.html

|  |
| --- |
| <div class="name">  <h1>  {{ surname|title }} {{ name|title }} {{ middle\_name|title }}  </h1>  </div> |

Шаблонные фильтры указываются через вертикальную черту (|). В примере используется фильтр title, который преобразует первый символ слов в верхний регистр, остальные в нижний.

Полный перечень [встроенных шаблонных фильтров](http://djbook.ru/rel1.8/ref/templates/builtins.html#ref-templates-builtins-filters). Если вы не найдете подходящего вам ильтра, довольно просто можно создать свой собственный. По сути, шаблонный фильтр, это просто функция Python (подробнее [тут](http://djbook.ru/rel1.4/howto/custom-template-tags.html)).

Теория:Модели + ORM = данные

Краткая теория баз данных. Введение в django-ORM. Подключение и создание базы данных. Несколько слов о миграциях. Работа с данными через Модель.

[Краткая теория баз данных(БД)](#_c713xnuw3xem)

[Введение в Django ORM](#_qgye5r7e5sog)

[Подключение БД](#_nf8xqspq9y8a)

[Выбор БД](#_bagr4g6fvkps)

[Установка и настройка](#_n9kaqrm7t2tx)

[Работа через консоль](#_4rk73dktvr8u)

[Добавим супер-пользователя](#_kuqxjcyn63vr)

[Запускаем интерактивную консоль django](#_oady015qnxh)

[Разбираем подробнее получившееся](#_30em3shr3tn5)

[Несколько слов о миграциях](#_b6jjfgmvrksv)

[Создаем собственную модель](#_hg2all4hcff9)

[Определяем поля](#_dcmgmacqnip)

[Определяем типы полей](#_a0levpxip9ru)

[Создаем модель](#_j4e0mq9dvffb)

[Создадим таблицу в БД](#_7505r6hnwqnc)

[Создаем объекты и сохраняем в БД](#_kpttjfr3tsig)

[Отображаем данные из базы на html-странице](#_c79xrfx887gw)

# 

# Краткая теория баз данных(БД)

База данных - набор сведений, хранящихся некоторым упорядоченным способом. Можно сравнить базу данных со шкафом, в котором хранятся документы. Иными словами, база данных - это хранилище данных. Сами по себе базы данных не представляли бы интереса, если бы не было систем управления базами данных (СУБД).

Система управления базами данных - это совокупность языковых и программных средств, которая осуществляет доступ к данным, позволяет их создавать, менять и удалять, обеспечивает безопасность данных и т.д. В общем СУБД - это система, позволяющая создавать базы данных и манипулировать сведениями из них. А осуществляет этот доступ к данным СУБД посредством специального языка - SQL.

SQL - язык структурированных запросов, основной задачей которого является предоставление простого способа считывания и записи информации в базу данных.

Создавая базу данных, мы стремимся упорядочить информацию по различным признакам для того, чтобы потом извлекать из нее необходимые нам данные в любом сочетании. Сделать это возможно, только если данные структурированы. Структурирование - это набор соглашений о способах представления данных. Понятно, что структурировать информацию можно по-разному. В зависимости от структуры различают иерархическую, сетевую, реляционную, объектно-ориентированную и гибридную модели баз данных. Самой популярной на сегодняшний день является реляционная структура.

Если говорить упрощенно, базы данных представляют собой связанные таблицы. Каждая из таблиц содержит информацию об объектах одного типа. Из названий таблиц становиться понятно, что данные в каждой таблице принадлежат одной и той же группе объектов. Каждая строка в этих таблицах однозначно определяет один объект из соответствующей группы. Вообще, база данных может состоять из одной или нескольких таблиц. Запись, в свою очередь, состоит из нескольких полей, каждое из которых содержит элемент данных об объекте.

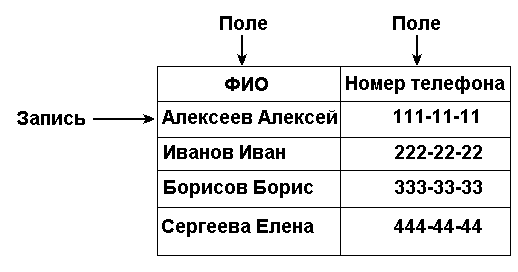


рис. пример таблицы

Современные базы данных - очень мощные инструменты, позволяющие осуществлять очень гибкую выборку данных, хранить и обрабатывать огромные массивы, следящие за целостностью.

# Введение в Django ORM

Django поддерживает четыре системы управления базами данных:

PostgreSQL (<http://www.postgresql.org> ).

SQLite 3 ([http://www.sqlite.org](http://www.sqlite.org/)).

MySQL ([http://www.mysql.com](http://www.mysql.com/)).

Oracle (<http://www.oracle.com> ).

По большей части движки для перечисленных баз данных хорошо работают с ядром среды Django.

Django предоставляет абстрактный уровень (“модели”) для создания структуры и управления данными вашего приложения, что позволяет абстрагироваться от особенностей реализации конкретной БД и писать запросы на более высоком(объектном) уровне. Знание SQL не будет лишним, но и без его знания можно успешно писать довольно сложные и эффективные запросы.

ORM (англ. Object-Relational Mapping, рус. объектно-реляционное отображение) — технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных».

Модели отображают информацию о данных, с которыми вы работаете. Они содержат поля и поведение ваших данных. Обычно одна модель представляет одну таблицу в базе данных.

Вот пример модели, которая определяет гипотетического человека(Person), с именем(first\_name) и фамилией(last\_name):

|  |
| --- |
| from django.db import models  class Person(models.Model):  first\_name = models.CharField(max\_length=30)  last\_name = models.CharField(max\_length=30) |

first\_name и last\_name fields модели. Каждое поле определено как атрибут класса, и каждый атрибут соответствует полю таблицы в базе данных.

Самая важная часть модели – и единственная обязательная – это список полей таблицы базы данных которые она представляет. Поля определены атрибутами класса. Тип поля показывает, в каком виде ваши данные будут храниться в базе, и что важнее, в качестве каких типов данных будут доступны при обращении к ним через атрибуты объекта. Так же тип поля определяет валидацию данных, перед сохранение в БД.

Валидация - проверка данных на корректность.

Еще раз, модель является посредником между данными в БД и данными в виде python-объекта. Вы создаете объект, его атрибутам присваиваете значения, а при сохранении в БД, модель транслирует ваш python-объект в SQL запрос. И наоборот, при чтении данных, модель формирует SQL запрос и преобразует его в python-объект/список python-объектов.

Кажется сложным? Пара примеров и все станет на свои места.

# Подключение БД

Прежде чем начать пользоваться базой, ее нужно подключить.

## Выбор БД

Несмотря на то, что sqlite3 являет наиболее медленной базой, и практически не используется в высоконагруженных проектах, мы будем использовать именно ее. Причина проста - ее вообще не нужно ни устанавливать, ни настраивать, она по умолчанию доступна в любой ОС. При необходимости, вы всегда можете перейти к любой БД, прочитав способ подключения. И помните, переход к другой БД совершенно не затронет код работы с базами.

Еще одним важным преимуществом(для начинающих django-программистов) является то, что sqlite3 представляет собой просто файл(с именем db.sqlite3 по умолчанию), и если что-то напортачите( а в начале вы часто будете ошибаться) можно просто удалить данный файл и попробовать заново.

## Установка и настройка

В Django 1.8 sqlite3 подключена по умолчанию, проверить это можно, зайдя в файл settings.py и увидев строки

|  |
| --- |
| DATABASES = {  'default': {  'ENGINE': 'django.db.backends.sqlite3',  'NAME': os.path.join(BASE\_DIR, 'db.sqlite3'),  }  } |

или поменяв на таковые, если в вашей версии значения проекта по умолчанию другие

Чтобы создать файл с пустой базой данных, выполните команду для автоматического создания таблиц базы данных:

python manage.py migrate

При успешном выполнении появится файл db.sqlite3 в корне проекта.

## Работа через консоль

Важно научиться работать с ORM через консоль. Почему? Во-первых, так вы быстро сможете добавлять данные в базу и читать из базы, без необходимости создавать формы и вьюшки(функции-представления). Во-вторых вы полностью будете видеть весь процесс работы с ORM. И в-третьих, рано или поздно вам все равно придется этому научиться.

Приступим.

Самый быстрый и простой способ добавить данные в базу, это создать суперпользователя(администратора). Да, модель для пользователей уже существует в недрах Django. Подробно про эту модель и его поля [тут](http://djbook.ru/rel1.8/ref/contrib/auth.html#django.contrib.auth.models.User).

### Добавим супер-пользователя

python manage.py createsuperuser

Далее укажите логин, емэйл и пароль(если забудете пароль, не беда, его всегда можно сменить).

### Запускаем интерактивную консоль django

python3 manage.py shell

Чтобы использовать модель User, импортируем ее:

from django.contrib.auth.models import User

Выполняем САМЫЙ ПЕРВЫЙ запрос:

User.objects.all()

Получаем что-то типа:

>> [<User: booblegum>]

Создаем еще одного пользователя:

user = User.objects.create\_user('john', 'lennon@thebeatles.com', 'johnpassword')

Сохраняем его в БД:

user.save()

Получаем всех пользователей в базе:

User.objects.all()

Получаем что-то типа:

>> [<User: booblegum>, <User: john>]

### Разбираем подробнее получившееся

Рассмотрим запрос: User.objects.all()

User - это модель. objects - менеджер объекта. Каждая модель содержит минимум один менеджер и называется он objects. Метод .all() менеджера ищет в базе все записи на класс User и возвращает их в виде списка python-объектов. Подробнее [тут](http://djbook.ru/rel1.7/topics/db/queries.html#making-queries).

Поэкспериментируйте со списком полученных объектов, например у User’ов есть атрибуты email и username. Вы же знаете как получить значение атрибута?

Теперь рассмотрим создание пользователя:

Чтобы создать запись в БД, нужно:

1. Создать модель для данного объекта (у нас уже есть модель User)
2. Создать экземпляр класса модели и заполнить обязательные поля(это мы выполнили командой user = User.objects.create\_user('john', 'lennon@thebeatles.com', 'johnpassword'))
3. У экземпляра класса вызвать метод save()

# Несколько слов о миграциях

Прежде чем приступить к созданию собственных моделей, нужно немного сказать о миграциях, т.к. с версии django 1.7 они стали неотъемлемой частью фрэймворка.

Django использует миграции для переноса изменений в моделях (добавление поля, удаление модели и т.д.) на структуру базы данных. Миграции создавались в основном для автоматической работы, но вам необходимо знать когда их создавать, запускать и как решать различные проблемы.

До версии 1.7, Django позволял только добавлять новые модели в базу данных; не было возможности изменять или удалять существующие модели, используя команду syncdb (предок команды migrate).

Сторонние инструменты, самый известный South, позволяли создавать и выполнять миграции. Со временем было решено перенести этот функционал в Django.

Django предоставляет две команды для работы с миграциями и структурой базы данных:

migrate, которая отвечает за применение миграций, за откат миграций и за вывод статуса миграций.

makemigrations, которая отвечает за создание новых миграций на основе изменений в моделях.

sqlmigrate, которая выводит SQL запросы для миграции.

Стоит отметить, что миграции создаются и работают в контексте отдельного приложения. В частности, можно создать приложение, которое не использует миграции - такие приложения имитируют старое поведение и просто создают новые модели.

Следует рассматривать миграции, как систему контроля версий для базы данных. makemigrations отвечает за сохранение состояния моделей в файле миграции - аналог коммита - а migrate отвечает за их применение к базе данных.

Файлы с миграциями находятся в каталоге “migrations” приложения. Они являются частью приложения и должны распространяться вместе с остальным кодом приложения. Они должны создаваться при разработке и потом применяется на машинах коллег, тестовом и “боевом” серверах.

Миграции работают идентично на одном и том же наборе данных. Это означает, что на сервере разработки, тестовом и “боевом” серверах вы получите один и тот же результат при одинаковых условиях выполнения миграций.

Django создаст миграции при любых изменениях модели или полей - даже тех параметров, которые не влияют на базу данных - т.к. единственный способ восстановить состояние моделей - это хранить все изменения в истории. Вам могут понадобится эти параметры в миграциях данных в будущем (например, если вы добавите собственную проверку данных).

Подробнее [тут](http://djbook.ru/rel1.8/topics/migrations.html).

# Создаем собственную модель

В рамках работы над нашим мини-проектом About ME, создадим модель места работы.

Определим поля места работы(возьмем за отправную точку список полей на hh.ru).

## Определяем поля

* Организация (в которой работали)
* Регион (нахождения организации)
* Сайт (понятно чей -) )
* Должность
* Обязанности функции
* Время работы (полных месяцев)

Пока полей достаточно.

Далее определяем типы полей, которые в свою очередь, определят проверку и типы хранимых данных.

[Тут](http://djbook.ru/rel1.7/ref/models/fields.html#field-types) вы найдете полное описание всех типов полей. Если вы впервые работаете с моделями, рекомендуется изучить все доступные типы и подбирать наиболее подходящие.

## Определяем типы полей

* Организация (CharField - название организации будем хранить в виде обычного текста)
* Регион (CharField)
* Сайт (CharField)
* Должность(CharField)
* Обязанности функции (TextField - описание будет в произвольном виде, этот тип наиболее подходит)
* Время работы (PositiveIntegerField - целое значение, с доп.защитой от отрицательных значений)

## Создаем модель

models.py

|  |
| --- |
| from django.db import models  class Work(models.Model):  organization = models.CharField(verbose\_name='Организация', max\_length=32)  region = models.CharField(verbose\_name='Регион', max\_length=32, blank=True)  site = models.CharField(verbose\_name='Сайт', max\_length=64, blank=True)  position = models.CharField(verbose\_name='Должность', max\_length=16)  duties = models.TextField(verbose\_name='Обязанности')  period = models.PositiveIntegerField(verbose\_name='Время работы', default=1) |

Обратите внимание: поля типа CharField имеют параметры поля max\_length - ограничивающие максимальное кол-во символов, это значение используется для валидации(проверки) данных перед сохранением в БД. Поля, имеющие параметр blank=True, являются необязательными при заполнении формы, если поле не заполнено, в базу сохраняется пустая строка. verbose\_name - отображаемое имя поля, будет использоваться при использовании форм. default - значение по умолчанию. Полный список параметров полей см. [тут](http://djbook.ru/rel1.7/ref/models/fields.html#module-django.db.models.fields).

## Создадим таблицу в БД

Объекты (в данном случае наши Работы) будут сохраняться в БД, а для этого им нужна таблица, построенная на основании нашей модели, для этого:

1. Добавить наше приложение в settings.py

settings.py (добавить)

|  |
| --- |
| INSTALLED\_APPS = (  'django.contrib.admin',  'django.contrib.auth',  'django.contrib.contenttypes',  'django.contrib.sessions',  'django.contrib.messages',  'django.contrib.staticfiles',  'mainapp'  ) |

Это необходимо, чтобу миграции знали где искать models.py

1. Создать миграцию

manage.py makemigrations

В ответ получите что-то типа:

>> Migrations for 'mainapp':  
 0001\_initial.py:  
 - Create model Work

1. Применить миграцию (добавить таблицу в БД)

python3 manage.py migrate mainapp

## Создаем объекты и сохраняем в БД

1. Создаем объект “Работа”, через консоль

from mainapp.models import Work  
work = Work()  
work.organization = ‘Test org’  
work.site = ‘test.ru’  
work.duties = ‘Тестовое описание...’  
work.save()

1. Получаем список работ из базы, проверяем значения атрибутов:

works = Work.objects.all()  
works[0].organization  
works[0].duties  
works[0].period

Мини-задание: Самостоятельно создайте несколько объектов “Работа”, попробуйте различные вариант задания значений для полей, поэкспериментируйте с результатами.

# Отображаем данные из базы на html-странице

Теперь наши данные хранятся в полноценной БД. Используя те же команды, что мы использовали в консоли, мы можем получить нужные нам данные во views.py и используя шаблон, отобразить на нашей странице

views.py

|  |
| --- |
| from .models import Work  def works(request):  place\_of\_work = Work.objects.all()  return render\_to\_response("works.html", {'place\_of\_work':  [work.organization for work in place\_of\_work]}) |

Подумайте и скажи, тем плоха текущая реализация?

Думайте… (ответ ниже)

Ответ: Мы снова обрабатываем отображение в функции-представлении, гораздо правильнее эту задачу получить шаблону, передав ему полные объекты. Но это уже ваша задача.

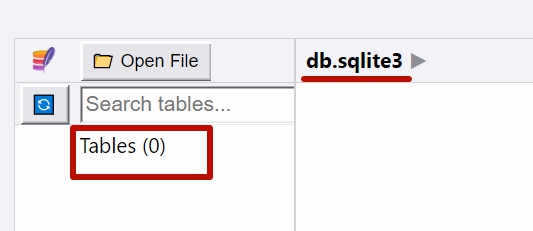
Инструкция: работа с БД

## Step-0 “Знакомимся с программой просмотра БД”

Заходим на [сайт](https://sqliteviewer.app/).

С помощью этого сервиса смотрим содержимое файла **db.sqlite3**(файл в корне вашего проекта)

*Выберите "Открыть базу данных", укажите на файл БД*

Убеждаемся что наша БД пустая(в ней нет ни единой таблицы):  


## Step-1 “Применяем миграцию”

Применяем миграцию(создаем структуру БД): python manage.py migrate

Обновляем содержимое БД: 🔄

Обратите внимание: Уже отображается 10 таблиц, хотя вы еще ничего не создавали. Это таблицы, которые Django создает для своих внутренних механизмов. Но все таблица пока пустые, т.е. в них нет данных.

## Step-2 “Создаем модель для товара”

Код модели(**models.py**):

|  |
| --- |
| **class** Item(models.Model):  name = models.CharField(max\_length=100)  brand = models.CharField(max\_length=100)  count = models.PositiveIntegerField() |

Все типы полей [тут.](https://docs.djangoproject.com/en/4.1/ref/models/fields/#field-types)

Параметры полей [тут](https://docs.djangoproject.com/en/4.1/ref/models/fields/#field-options). *например* ***max\_length*** *- параметр, задающую макс длину.*

Слушаем объяснения преподавателя и понимаем как все это работает.

Создаем миграцию: python manage.py makemigrations

Применяем миграцию к БД, знакомой командой: python manage.py migrate

Подробнее про создание моделей(**fix**).

## Step-3 “Устанавливаем django-extensions и ipython”

Находясь в venv нашего проекта:  
$ pip install django-extensions

в settings.py дописываем:

|  |
| --- |
| **INSTALLED\_APPS** = [  **...**  **'django\_extensions'**  ] |

Ставим ipython:

$ pip install ipython

## Step-4 “Делаем записи в БД”

Запускаем консоль используя django\_extensions:   
python manage.py shell\_plus --ipython

Создаем объект используя модель:

|  |
| --- |
| In [x]: item = Item(name=**"Кроссовки"**, brand=**"abibas"**, count=10) |

Сохраняем объект в базе:

|  |
| --- |
| In [x]: item.save() |

**Обратите внимание** In [x]: не нужно набирать в консоли. Этот префикс напоминание, что мы выполняем команды в режиме интерактивного интерпретатора ipython.

## Step-5 “Проверяем наличие объекта в БД”

Повторно откройте файл **db.sqlite3**

Ищем таблицу с нашим объектом Item, проверяем, появилась ли соответствующая запись.

## Step-6 “Получение объектов из БД”

Для получения объектов из БД, делам запрос используя ORM:

|  |
| --- |
| In [x]: items = Item.objects.all()  In [x]: items |

Видим QuerySet наших объектов. Пока вы можете воспринимать объект QuerySet как аналог списка.

Посмотрим параметры нашего товара:

|  |
| --- |
| In [x]: items[0].name  In [x]: items[0].brand |

Инструкция: работа с формами

1. Создаем форму из модели  
   **forms.py**

|  |
| --- |
| from django.forms import ModelForm  from MainApp.models import Snippet  class SnippetForm(ModelForm):  class Meta:  model = Snippet  *# Описываем поля, которые будем заполнять в форме*  fields = [**'name'**, **'lang'**, **'code'**] |

1. Создание формы во view:

|  |
| --- |
| from MainApp.forms import SnippetForm  def create\_snippet(request):  form = SnippetForm()  return render(request, **'add\_snippet.html'**, {**'form'**: form}) |

1. Отображение формы в шаблоне  
   **add\_snippet.html**

|  |
| --- |
| <form action="url" method="post">  {% csrf\_token %}  <table>  {{ form }}  </table>  <input type="submit" value="Submit"/>  </form> |

1. Сохранение объекта при получении данных из формы  
   views.py:

|  |
| --- |
| def create\_snippet(request):  if request.method == **"POST"**:  form = SnippetForm(request.POST)  if form.is\_valid():  form.save()  return redirect(**"redirect\_url"**)  return render(request,**'add\_snippet.html'**,{**'form'**: form}) |

1. Полный метод обработки GET и POST запросов  
   Если запрос GET-отправляем html-страницу с формой, если POST - значит пришли данные от формы:

|  |
| --- |
| def add\_snippet\_page(request):  if request.method == "GET":  form = SnippetForm()  context = {  'pagename': 'Добавление нового сниппета',  'form': form  }  return render(request, 'pages/add\_snippet.html', context)  if request.method == "POST":  form = SnippetForm(request.POST)  if form.is\_valid():  form.save()  return redirect("snippets-list")  return render(request, 'add\_snippet.html', {'form': form}) |

1. Подробнее [про формы](https://djbook.ru/rel1.9/topics/forms/modelforms.html#module-django.forms.models). И [тут](https://developer.mozilla.org/ru/docs/Learn/Server-side/Django/Forms).

## Немного про кастомные формы

1. Хотим нестандартную форму со своим стилем оформления, например такую:

|  |
| --- |
| <form action="" method="post">  <div class="form-group row">  <div class="col-sm-8">  <input type="text" class="form-control form-control-lg" placeholder="Название сниппета">  </div>  </div>  <div class="form-group row">  <div class="col-sm-8">  <div class="dropdown">  <button class="btn btn-secondary dropdown-toggle" type="button" id="dropdownMenuButton"  data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">  Язык сниппета  </button>  <div class="dropdown-menu" aria-labelledby="dropdownMenuButton">  <a class="dropdown-item" href="#">Python</a>  <a class="dropdown-item" href="#">JavaScript</a>  <a class="dropdown-item" href="#">C++</a>  </div>  </div>  </div>  </div>  <div class="form-group row">  <div class="col">  <textarea cols="96" rows="10" placeholder="Код сниппета"></textarea>  </div>  </div>  <div class="form-group row">  <div class="col-8">  <input type="submit" class="btn btn-primary btn-block" value="Сохранить">  </div>  </div>  </form> |

1. В классе формы можем изменить как виджет(html-элемент для отображения нашего поля на странице), так и прочие параметры виджета.

|  |
| --- |
| class SnippetForm(ModelForm):  class Meta:  model = Snippet  *# Описываем поля, которые будем заполнять в форме*  fields = [**'name'**, **'lang'**, **'code'**]  widgets = {  **'name'**: TextInput(attrs={**"class"**:**"form-control form-control-lg"**, **'placeholder'**: **'Название сниппета'**}),  }  labels = {  **'name'**: **''**  } |

В свойстве widgets описываем словарь, где ключами являются наши поля, а значениями виджеты-отображения.

В свойстве label словарь отображаемых меток(подписей) виджетам

1. В шаблоне с кастомной формой делаем подстановки каждого поля отдельно:

|  |
| --- |
| <form action="{% url 'snippet\_add' %}" method="POST">  {% csrf\_token %}  <div class="my-class">  {{form.name}}  </div>  <div class="my-class">  {{form.lang}}  </div>  <button type="submit">Create</button>  </form> |

1. Подробнее [тут](https://docs.djangoproject.com/en/dev/topics/forms/modelforms/#overriding-the-default-fields).

Инструкция: подключение статики

## Step-1 "create static dir"

Создаем директорию static в корне проекта - в этой папке должны располагаться все файлы статики (css, картинки).

## Step-2 “settings”

В **settings.py** добавить(в конец файла):

|  |
| --- |
| STATICFILES\_DIRS = [  BASE\_DIR / **"static"**  ] |

## Step-3 “urls”

В **urls.py** добавить(выделенное зеленым):

|  |
| --- |
| from django.conf import settings  from django.conf.urls.static import static  urlpatterns = [  path(**'admin/'**, admin.site.urls),  ]+ static(settings.STATIC\_URL, document\_root=settings.STATIC\_ROOT) |

## Step-4 “template”

В шаблоне вставляем:

|  |
| --- |
| **{% load static %}**  <link rel="stylesheet" href="{% **static** 'style.css' %}"> |

**{% load static %}** - рекомендуется вставлять в начале html-документа   
Или аналогично в места вставки статики

Пример вставки изображения:

|  |
| --- |
| <img src="{% **static** 'img/logo.png' %}" alt=""> |

где '**img/logo.png**' - относительный путь до картинки внутри папки static

**{% load static %}** - вставляется в начале каждого html файла в котором используется {% static %} и только один раз!

Кратко про bootstrap

**Bootstrap** — это открытый и бесплатный HTML, CSS и JS фреймворк, который используется веб-разработчиками для быстрой верстки адаптивных дизайнов сайтов и веб-приложений.

* [Подключение bootstrap](https://getbootstrap.com/docs/4.0/getting-started/introduction/#quick-start)
* [Grid system](https://getbootstrap.com/docs/4.0/layout/grid/)
* [Bootstrap navbar](https://getbootstrap.com/docs/4.0/components/navbar/#nav)

1. [официальная русская документация](https://bootstrap-4.ru/)
2. [Введение](https://bootstrap-4.ru/docs/4.5/getting-started/introduction/)
3. [Grid system](https://getbootstrap.com/docs/4.0/layout/grid/)

Инструкция: Авторизация пользователя

1. Создаем суперпользователя(администратора):  
   $ (venv) **python manage.py createsuperuser**
2. Создаем форму авторизации:

|  |
| --- |
| <form method="**post**" action="url">**{% csrf\_token %}**  <input name="username" type="text" placeholder="username">  <input name="password" type="password" placeholder="password">  <button type="submit">login</button>  </form> |

1. Связываем url c функцией-обработчиком  
   **urls.py:**

|  |
| --- |
| urlpatterns = [  ...  path(**'**url**/'**, views.**login\_page**, name=**'login'**),  ...  ] |

1. Принимаем данные формы в функцию-обработчик  
   **views.py:**

|  |
| --- |
| def **login\_page**(request):  if request.method == **'POST'**:  username = request.POST.get(**"username"**)  password = request.POST.get(**"password"**)  print(**"username ="**, username)  print(**"password ="**, password) |

1. Авторизация(залогинивание) пользователя

|  |
| --- |
| from django.contrib import auth  from django.shortcuts import redirect  ...  def **login\_page**(request):  if request.method == **'PO2ST'**:  username = request.POST.get(**"username"**)  password = request.POST.get(**"password"**)  *# print("username =", username)*  *# print("password =", password)*  user = auth.authenticate(request, username=username, password=password)  if user is not None:  auth.login(request, user)  else:  *# Return error message*  pass  return redirect(**'home'**) |

Примечание: если хотите реализовать перенаправление на ту страницу, на которой вы логинились, замените return redirect(**'home'**) на:   
return redirect(request.META.get('HTTP\_REFERER', '/'))

1. Проверка на авторизацию пользователя в шаблоне:

|  |
| --- |
| {% **if** **request**.**user**.is\_authenticated %}  Welcome, {{**request**.**user**.username }}  {% **else** %}  <тут форма авторизации>  {% **endif** %} |

1. Разлогинивание пользователя во view.py:

|  |
| --- |
| auth.logout(request) |

1. Подробнее про [систему авторизации(rus)](https://django.fun/ru/docs/django/4.0/topics/auth/default/#using-the-django-authentication-system), [система авторизации(eng)](https://docs.djangoproject.com/en/4.1/topics/auth/).
2. Красивая html-форма авторизации:

|  |
| --- |
| <div class="navbar-nav mr-right nav-item dropdown dropdown-menu-right">  <a class="btn btn-outline-primary my-2 my-sm-0 dropdown-toggle" href="#" id="navbarDropdown"  role="button" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">Авторизация</a>  <div class="dropdown-menu dropdown-menu-right" aria-labelledby="navbarDropdown">  <form class="w-100 px-4 py-3" action="" method="">  <div class="form-group form-inline">  <input type="text" class="form-control" placeholder="Username" required>  </div>  <div class="form-group form-inline">  <input type="password" class="form-control" required>  </div>  <button type="submit" class="btn btn-primary btn-block">Войти</button>  </form>  <div class="dropdown-divider"></div>  </div>  </div> |

Форму ставьте в <header><nav>..сюда..</nav</header>

**Важно!** проверьте форму и ее виджеты, если чего-то не хватает - добавьте!

1. Поле user для модели Сниппета:

|  |
| --- |
| from django.contrib.auth.models import User  ...  user = models.ForeignKey(to=User, on\_delete=models.CASCADE,  blank=True, null=True) |

1. Форма регистрации пользователя:

|  |
| --- |
| from django.contrib.auth.models import User  from django.forms import CharField, PasswordInput  from django.core.exceptions import ValidationError  class UserRegistrationForm(ModelForm):  class Meta:  model = User  fields = [**"username"**, **"email"**]  password1 = CharField(label=**"password"**, widget=PasswordInput)  password2 = CharField(label=**"password confirm"**, widget=PasswordInput)  def clean\_password2(self):  pass1 = self.cleaned\_data.get(**"password1"**)  pass2 = self.cleaned\_data.get(**"password2"**)  if pass1 and pass2 and pass1 == pass2:  return pass2  raise ValidationError(**"Пароли не совпадают или пустые"**)  def save(self, commit=True):  user = super().save(commit=False)  user.set\_password(self.cleaned\_data[**"password1"**])  if commit:  user.save()  return user |

Подробнее можно почитать [тут](https://www.crunchydata.com/blog/building-a-user-registration-form-with-djangos-built-in-authentication).

1. Подстановка формы в шаблон:

|  |
| --- |
| <form action="" method="post">  {% csrf\_token %}  {{ form.as\_p }}  <input type="submit" value="Создать пользователя">  </form> |

1. Обработка данных формы и создание нового пользователя по аналогии с созданием сниппета.

Инструкция: комментарии

## Что хотим

Добавить функционал по добавлению комментариев к сниппету. Комментарии могут оставлять только авторизованные пользователи. Комментарии могут быть удалены администратором сайта и отредактированы автором.

## Реализация

1. Создаем модель для комментариев:

|  |
| --- |
| class Comment(models.Model):  text = ...  creation\_date = ...  author = ...  snippet = ... |

**author** - поле связывающее комментарий с моделью User  
**snippet** - связывает с моделью Snippet

1. Создаем форму:

|  |
| --- |
| class CommentForm(ModelForm):  class Meta:  model = Comment  fields = [**...**] |

1. Добавляем url для получения данных из формы создания комментария:

|  |
| --- |
| urlpatterns = [  ...  path(**'comment/add'**, views.comment\_add, name=**"comment\_add"**), |

1. Думаем: где разместим форму “добавление комментария”?  
   *Понимаем,* *что их лучше создать на “Странице сниппета” -)*
2. Добавляем форму в html-шаблон:

|  |
| --- |
| <form action="{% url 'comment\_add' %}" method="POST">  {{comment\_form}}  <button type="submit">Добавить</button>  </form> |

1. Добавляем функцию-обработчик:

|  |
| --- |
| def comment\_add(request):  if request.method == **"POST"**:  comment\_form = CommentForm(request.POST)  if comment\_form.is\_valid():  comment = comment\_form.save(commit=False)  comment.author = ...  comment.snippet = ...  comment.save()  return redirect(**f'/snippet/**{snippet\_id}**'**)  raise Http404 |

1. Создаем несколько комментариев для определенного сниппета.
2. Отображаем комментарии внизу страницы сниппета:

|  |
| --- |
| <table border="1">  <tr>  <th>Автор</th>  <th>Комментарий</th>  </tr>  {% for comment in comments %}  <tr>  <td>{{comment.author.username}}</td>  <td>{{comment.text}}</td>  </tr>  {% endfor %}  </table> |

Подробнее про [обратные связи в моделях](https://djbook.ru/rel1.6/ref/models/fields.html#django.db.models.ForeignKey.related_name).

1. Тестируем получившееся.

Теория: шаблонные фильтры

В django существует большой набор встроенных шаблонных тегов и фильтров, подробнее [о тегах](https://djbook.ru/rel1.5/ref/templates/builtins.html#built-in-tag-reference) и [фильтрах](https://djbook.ru/rel1.5/ref/templates/builtins.html#built-in-filter-reference).

Есть возможность создавать собственные шаблонные теги.

## Создание собственных шаблонных тегов

1. В папке с приложением(App) создайте папку **templatetags**
2. В **templatetags** создайте py-файл с вашими шаблонными тегами  
   Например: **my\_tags.py**
3. Приложение содержащее собственные теги и фильтры должно быть добавлено в INSTALLED\_APPS, что бы тег {% load %} мог загрузить его
4. И в шаблоне вы будете использовать:

{% load **my\_tags** %}

1. Важно понимать, что **шаблонный фильтр - это просто функция** питона, принимающая значение(к которому применяется фильтр, в качестве аргумента).  
   Пояснение:

|  |
| --- |
| {{ “**hello**” | **upper** }}  def **upper**(**value**):  return value.upper() |

Тут строка “hello” будет передана в функцию upper в переменную value.

1. Пример  
   **my\_tags.py**

|  |
| --- |
| from django import template  register = template.Library()  def **is\_empty**(value, alt):  if value:  return value  return alt  register.filter(**'is\_empty'**, is\_empty) |

**view\_snippets.html**

|  |
| --- |
| …  {% load my\_tags %}  …  <td>{{ snippet.user.username|**is\_empty**:"аноним" }}</td>  ... |

1. Подробнее [тут](https://djbook.ru/rel1.4/howto/custom-template-tags.html#custom-template-tags-and-filters).

Теория: загрузка изображений

1. Установить: pip install Pillow
2. В **settings.py** добавим:

|  |
| --- |
| ...  TEMPLATES = [  {  'BACKEND':...  'OPTIONS': {  'context\_processors': [  ...  'django.template.context\_processors.media',  ],  },  },  ]  ...  MEDIA\_URL = '/media/'  MEDIA\_ROOT = os.path.join(BASE\_DIR, 'media') |

Теперь все файлы будут загружаться в папку media, в корне проекта.  
Если данной папки нет, она будет создана автоматически, при загрузке первого изображения.

1. В urls.py:

|  |
| --- |
| urlpatterns = [  ...  ]+ static(...) +static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT) |

1. В модели добавьте поле для изображения:

|  |
| --- |
| image = models.ImageField(**upload\_to**=**"images"**) |

**upload\_to** - указывает доп.директорию в папке media, для загрузки изображения из данного поля

<https://docs.djangoproject.com/en/dev/ref/models/fields/#django.db.models.FileField>

<https://docs.djangoproject.com/en/dev/ref/models/fields/#django.db.models.ImageField>

1. Для отображения в шаблоне:

|  |
| --- |
| <img src="{{ MEDIA\_URL }}{{ **object**.**image** }}"> |

где **object** - ваш объект, в нашем случае **comment**

1. Для сохранения во view.py:

|  |
| --- |
| form = Form(request.POST, request.FILES) |

1. Чтобы форма умела передавать файлы на сервер, добавляем:

|  |
| --- |
| <**form** method="POST" enctype="multipart/form-data"> |

1. Добавить в форму отображение нового поля:

|  |
| --- |
| class CommentForm(ModelForm):  class Meta:  model = Comment  # Описываем поля, которые будем заполнять в форме  fields = ['text', "image"] |

Теория: админка Django

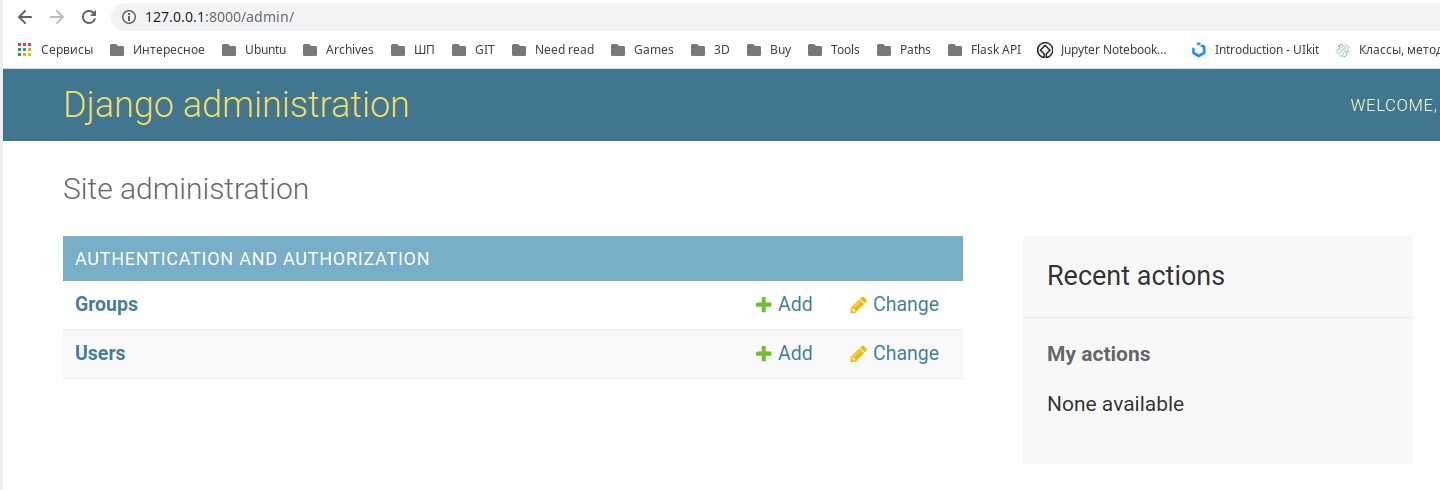
Вступление

Приложение Django admin может использовать ваши модели для автоматического создания части сайта, предназначенной для создания, просмотра, обновления и удаления записей. Это может сэкономить вам много времени в процессе разработки, упрощая тестирование ваших моделей на предмет правильности данных.

## Использование встроенной админки django

1. В **urls.py** восстанавливаем:

|  |
| --- |
| **...**  **from django.contrib import admin**  **urlpatterns = [**  **...**  **path('admin/', admin.site.urls),** |

1. Создаем суперпользователя, если еще не создан:   
   python manage.py createsuperuser
2. Заходим по url: /admin/  
   
3. Чтобы иметь возможность редактировать другие модели, регистрируем из в **admin.py**

|  |
| --- |
| from django.contrib import admin  from MainApp.models import Snippet  admin.site.register(Snippet) |

1. Подробнее [тут](https://developer.mozilla.org/ru/docs/Learn/Server-side/Django/Admin_site) и [тут](https://djbook.ru/rel1.9/ref/contrib/admin/index.html).

Дополнительные материалы

Вступление

В данном документе приводятся ссылки на доп.материалы, использование которых дам вам полезные навыки и позволит доработать проект Snippets.

## Синтаксическая подсветка кода

Чтобы реализовать отображение кода в сниппетах с форматированием и синтаксической подсветкой используйте библиотеку [pygments](https://pygments.org/).   
[Тут](http://magazineit.ru/articles/web-programmirovanie/38/) понятный пример установки и использования на русском языке.

## Информационные сообщения

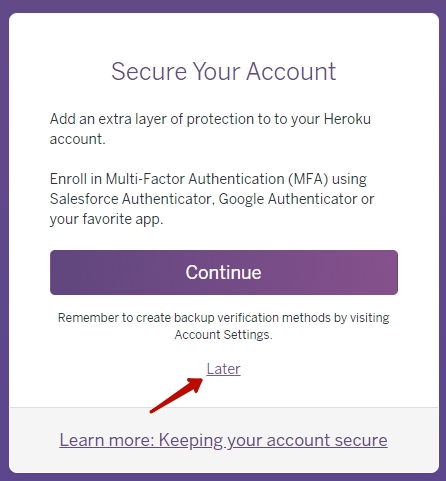
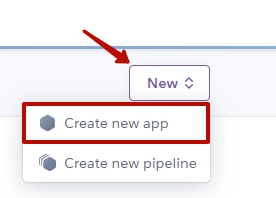
Когда пользователь выполняет какие-то действия на сайт: авторизуется, создает или удаляет сниппеты было бы неплохо выводить ему информационные сообщение, информируя о результатах его действий. Конечно можно каждый раз перенаправлять на страницу с соответствующим сообщением, но гораздо лучше использовать механизм [django-messages](https://docs.djangoproject.com/en/3.1/ref/contrib/messages/)(eng).

Или [тут](https://pocoz.gitbooks.io/django-v-primerah/content/glava-4-sozdanie-social-website/registratsiya-polzovatelei-i-profili-polzovatelei/ispolzovanie-messages-framework.html) простой пример использования rus.

Инструкция: развертывание Django проекта на Heroku

1. Регистрируемся [на heroku](https://signup.heroku.com/login). Указываем Role: hobbies и language: python  
   Важно: логинимся в приложении heroku внутри виртуальной машины

***Рекомендация***: по возможности используйте gmail - почту. На mail письма с подтверждением могут идти долго.

1. Дожидаемся письмо на почту и подтверждаем регистрацию  
   Получив такое сообщение:  
      
   Выбираем Later
2. Создаем новое приложение “Create new app”  
   
3. Устанавливаем консольное приложение heroku:   
   $ sudo snap install --classic heroku

Для WSL2: curl https://cli-assets.heroku.com/install.sh | sh

1. Логинимся: $ heroku login

*Убедитесь, что отключили VPN*

1. Проект уже должен быть по git-ом.
2. Делаем коммит с последними изменениями: $ git commit -m “Comment”
3. В локальный репозиторий добавляем ссылку на удаленный репозиторий heroku: $ heroku git:remote -a you\_app\_name

## 

## Вносим изменения в проект, для правильного развертывания на heroku

1. Создаем в корне проекта **Procfile**:

|  |
| --- |
| release: python manage.py collectstatic --noinput  web: gunicorn Snippets.wsgi --log-file - |

1. В settings.py дописываем:

|  |
| --- |
| import dj\_database\_url  db\_from\_env = dj\_database\_url.config(conn\_max\_age=500)  …  DATABASES = {  **'default'**: {  **'ENGINE'**: **'django.db.backends.sqlite3'**,  **'NAME'**: BASE\_DIR / **'db.sqlite3'**,  }  }  DATABASES[**'default'**].update(db\_from\_env)  ...  STATIC\_ROOT = BASE\_DIR / "staticfiles" |

Устанавливаем: $ pip install dj-database-url  
Подробнее про dj-database-url [тут](https://pypi.org/project/dj-database-url/).

1. Актуализируем **requirements.txt**В **requirements.txt** добавляем:

|  |
| --- |
| ***ipython***  gunicorn  django-heroku  psycopg2-binary |

***Для ipython - убираем версию***.

1. Указываем версию python  
   В корне проекта создаем файл **runtime.txt**:

|  |
| --- |
| **python-3.9.14** |

Если не указать версию, то по умолчанию будет использована 3.10.7, а с этой версией не работает библиотека backports.zoneinfo, используемая в django.

[Поддерживаемые версии python на heroku](https://devcenter.heroku.com/articles/python-support#specifying-a-python-version)

1. …

## Продолжаем шаманить с heroku

1. Для работы с postgresql потребуется установить addon Heroku Postgres.  
   $ heroku addons:create heroku-postgresql:hobby-dev
2. Проверьте что все закоммичено!
3. Пушим наш репозиторий на heroku: **$ git push heroku master**
4. Запускаем сайт с heroku
5. Тестируем...
6. Фиксим ошибки:
   * **heroku log --tail** ← посмотреть лог ошибок на сервере
   * **heroku run bash** - подключение к удаленному серверу
7. Проверяем работоспособность нашего сайт/проекта

## Дополнения

1. Если хотите запустить несколько скриптов по очереди, в **Procfile** пропишите их через символ **;**

|  |
| --- |
| release: <команда-1>**;** <команда-2>**;** <команда-3>  web: gunicorn Snippets.wsgi --log-file - |

1. Список хостов/доменов, для которых может работать текущий сайт

**settings.py:**

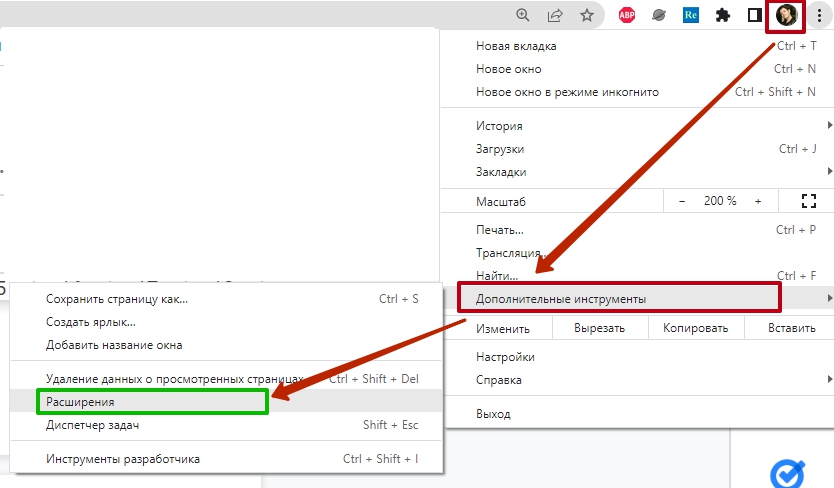
|  |
| --- |
| ALLOWED\_HOSTS = [**'\*'**] |

Или так, для heroku:

|  |
| --- |
| ALLOWED\_HOSTS = [  '.herokuapp.com', # Allow domain and subdomains  ] |

1. Обнуление/сброс репозитория отправленного на heroku:  
   $ heroku plugins:install heroku-repo  
   $ heroku repo:reset
2. Установка python для приложения heroku(если забыли установить при регистрации аккаунта):  
   $ heroku buildpacks:set heroku/python
3. …

## Проблемы с блокировкой

1. Заходим в расширения браузера  
   
2. Устанавливаем приложение для VPN, например, это:   
   [ZenMate](https://chrome.google.com/webstore/detail/free-vpn-zenmate-best-vpn/fdcgdnkidjaadafnichfpabhfomcebme?hl=ru)
3. Активируем VPN, выбираем Румыния (*или США, если Румыния не работает*)
4. Регистрируемся на heroku, указывая в качестве страны ту страну, которую выбрали в VPN
5. После регистрации, VPN нужно отключить

Доп.информация:   
настройка Django + Gunicorn + Nginx

Мы воспользовались сервисом heroku для развертывания проекта. На heroku уже настроены все нужные нам сервисы.

Если вы хотите научиться самостоятельно устанавливать и настраивать все сервисы:

* Gunicorn
* Nginx
* Postgres

То почитайте [эту статью](https://www.digitalocean.com/community/tutorials/how-to-set-up-django-with-postgres-nginx-and-gunicorn-on-ubuntu-18-04-ru).

# Работа со статикой

Думаю, рано или поздно, у вас возникнет проблема работы со статикой.

Веб-приложения обычно требуют различные дополнительные файлы для своей работы (изображения, CSS, Javascript и др.). В Django их принято называть “статическими файлами”(прим.пер. - или “статика”).

Вообще, в реальном “боевом” проекте django не занимается раздачей статики, это забота веб-сервера(nginx, apache и прочих), а пока в качестве такого веб-сервера выступает тестовый-джанго-сервер.

Пока не будем глубоко вникать в особенности работы со статикой и просто заставим сервер отдавать статические файлы при запросе браузера(да, именно браузер запрашивает эти файлы, встречая в html коде ссылки на них).

Создадим папку static в корне проекта для хранения всех статических файлов.

В конец settings.py Добавим код:

|  |
| --- |
| STATICFILES\_DIRS = (  BASE\_DIR / "static",  ) |

Обратите внимание: внимательно изучите глобальную переменную BASE\_DIR, это хороший способ получения абсолютного пути проекта, который я бы советовал использовать в прочих проектах, чтобы избежать путаницы с путями внутри самого проекта.

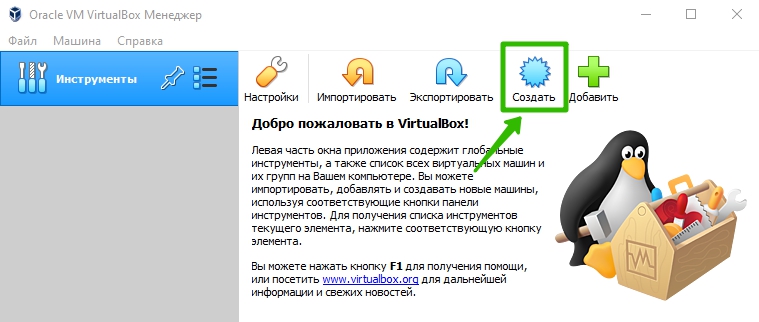
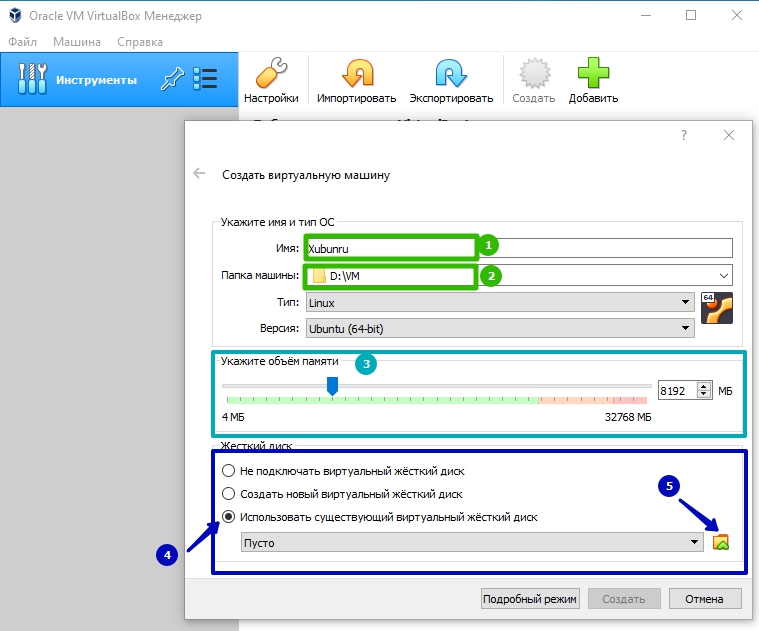
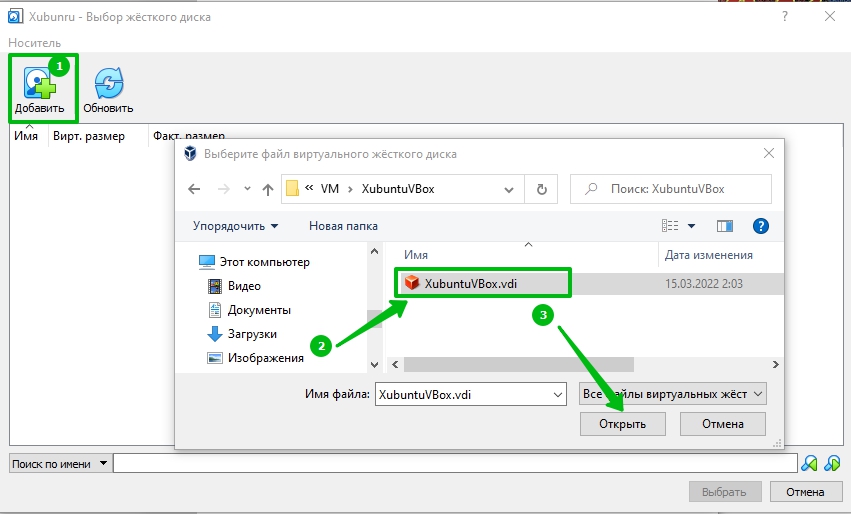
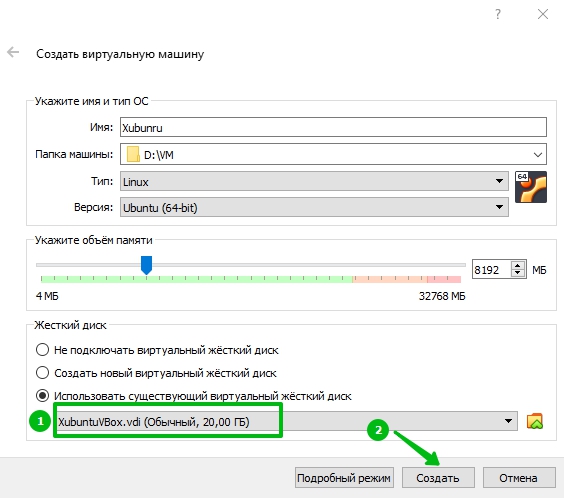
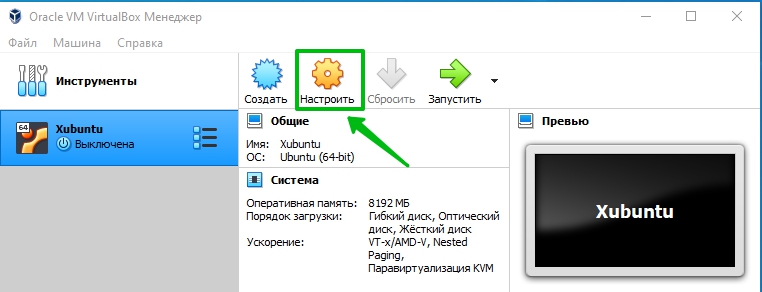
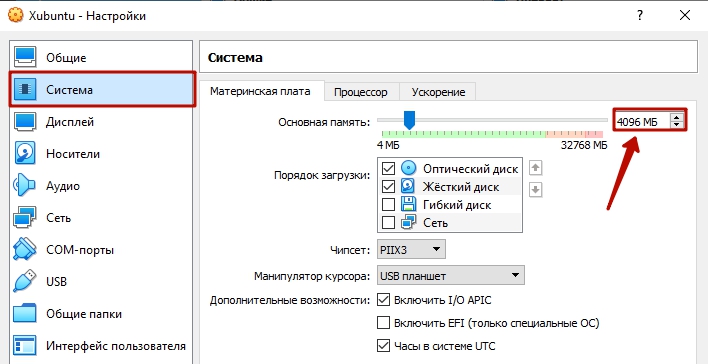
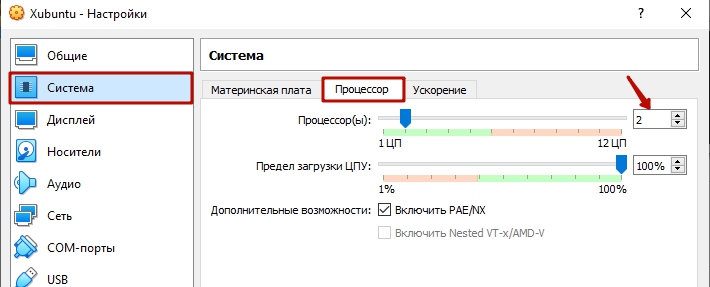
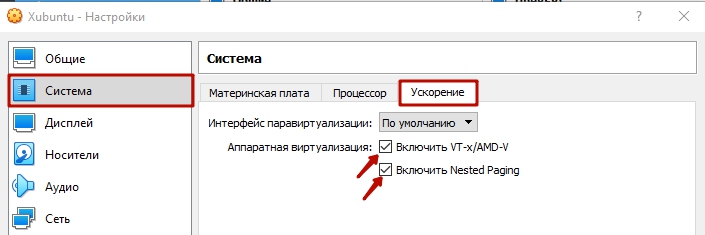
Вот и все, теперь статика отдается. Подробнее про работу со статическими файлами можете почитать [тут](https://docs.djangoproject.com/en/4.1/howto/static-files/#how-to-manage-static-files-e-g-images-javascript-css).

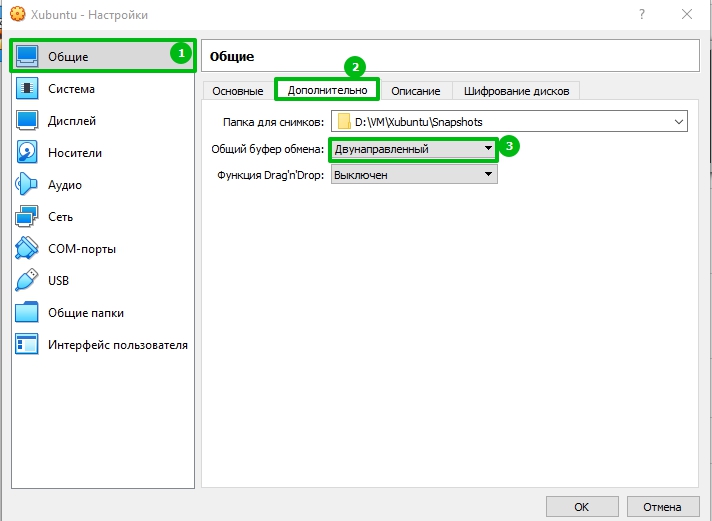
Инструкция: fixtures

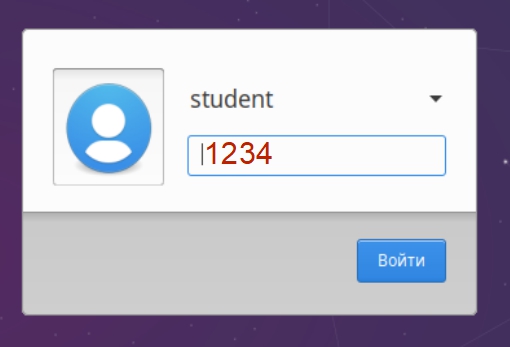
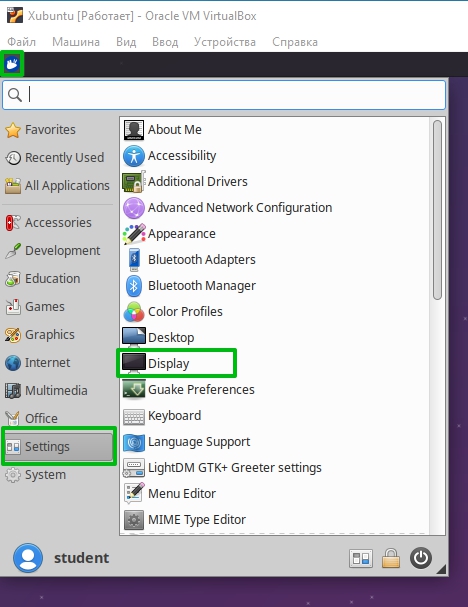
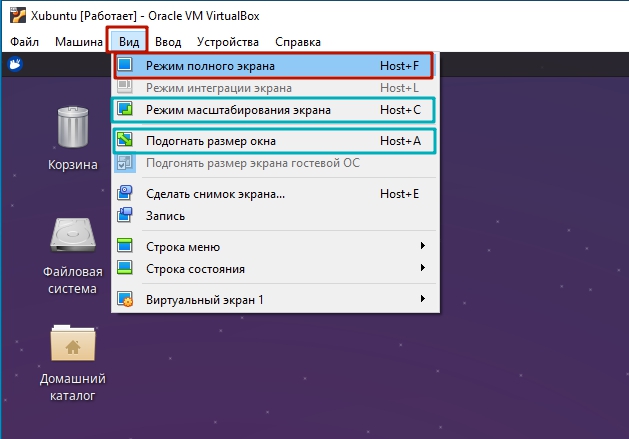
Подробнее про fixtures в django можно почитать [тут](https://realpython.com/django-pytest-fixtures/#fixtures-in-django).

1. Создайте папку **fixtures** в папке приложения(app)
2. Сохраните данные командой:   
   **python manage.py dumpdata MainApp** > MainApp/**fixtures**/save\_all.json  
   ,где **MainApp** - приложение(app), объекты которого вы хотите сохранить.
3. Загрузка данных из json-фикстуры:   
   **manage.py loaddata** <fixturename>,   
   где <fixturename> это имя созданного вами файла с данными

Настройка V-Box

1. Скачиваем VirtualBox с [офф-сайта](https://www.oracle.com/virtualization/technologies/vm/downloads/virtualbox-downloads.html)
2. Устанавливаем VirtualBox (cо всеми настройками по умолчанию)
3. Скачиваем образ виртуального диска: [**XubuntuVBox.rar**](https://drive.google.com/file/d/12Y0rZjZCLE0hCZye0Mt6w004vrwT8VWR/view?usp=sharing) (клик)
4. Распаковываем. Если нет архиватора, можно скачать [тут](https://www.7-zip.org/download.html). (Объем: **20Gb**)
5. Создаем виртуальную машину:  
   
6. Настраиваем создание:  
     
   1) Название вашей виртуальной машины. *Рекомендация*: Xubuntu  
   2) Папка для хранение ВМ (можно оставить ту, что указана по умолчанию)  
   3) Объем оперативной памяти выделяемой под ВМ (не более 50%)  
   4) Выбираем указанный пункт 5) Подключаем виртуальный диск
7. Добавляем виртуальный диск  
   
8. Жмем “Добавить”
9. Выбираем файл извлеченный из скачанного архива
10. Проверяем настройки:  
    
11. Тут появится выбранный виртуальный диск
12. Создаем виртуальную машину
13. Настройки виртуальный машины:  
      
    Проверяем, что выделили достаточно памяти (4-8Гб)  
      
      
    Пару ядер хватит. Но если у вас всего пара - то хватит и одного  
      
    Включаем аппаратную виртуализацию:  
      
    Если галочки не активны, значит аппаратная виртуализация или отсутствует(старый процессор) или(скорее всего) отключена в биосе. Тогда ищем [как включить аппаратную виртуализацию](https://www.google.com/search?q=%D0%B2%D0%BA%D0%BB%D1%8E%D1%87%D0%B8%D1%82%D1%8C+%D0%B0%D0%BF%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D0%BD%D1%83%D1%8E+%D0%B2%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8E+%D0%B2+%D0%B1%D0%B8%D0%BE%D1%81%D0%B5&oq=%D0%B2%D0%BA%D0%BB%D1%8E%D1%87%D0%B8%D1%82%D1%8C+%D0%B0%D0%BF%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D0%BD%D1%83%D1%8E+%D0%B2%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8E&aqs=chrome.1.69i57j0l3j0i22i30l4.6671j0j7&sourceid=chrome&ie=UTF-8), без нее все будет медленнее…

Включаем двусторонний буфер обмена

1. Запускаем VM(виртуальную машину). **Пароль**: **1234  
   **
2. Для удобной работы устанавливаем разрешение экрана:  
   
3. Выбираем удобный **ВИД**:  
     
   **Host** - по умолчанию **правый Ctrl**
4. ***Примечание***: в данной виртуальной машине установлены:
   1. Браузер Google Chrome
   2. IDE PyCharm
   3. IDE VsCode

Инструкция: WSL2

WSL@ - это подсистема Windows для Linux позволяет разработчикам запускать среду GNU/Linux с большинством программ командной строки, служебных программ и приложений непосредственно в Windows без каких-либо изменений и необходимости использовать традиционную виртуальную машину или двойную загрузку.

Официальная инструкция [тут](https://docs.microsoft.com/ru-ru/windows/wsl/install).

## Проверка требований для запуска WSL2

Для обновления до WSL 2 требуется Windows 10...

* Для систем x64: версия 1903 или более поздняя со сборкой 18362 или более поздней версии.
* Для систем ARM64: версия 2004 или более поздняя со сборкой 19041 или более поздней версии.
* Либо Windows 11.

Чтобы проверить версию и номер сборки, нажмите клавиши Windows+R, введите winver и нажмите кнопку ОК

Установка

## Установка WSL2

Официальная инструкция [тут](https://docs.microsoft.com/ru-ru/windows/wsl/install-manual#step-4---download-the-linux-kernel-update-package).

1. Выполните в терминале команду: wsl --install  
   По окончанию установки перезагрузите компьютер
2. Скачайте и установите пакет последней версии:  
   [Пакет обновления ядра Linux в WSL2](https://wslstorestorage.blob.core.windows.net/wslblob/wsl_update_x64.msi)
3. Запустите терминал(cmd) или powershell от имени администратора
4. Выполните в терминале команду: wsl --set-default-version 2

## Установка Ubuntu в WSL2

1. Запустите терминал(cmd) или powershell от имени администратора
2. Выполните: wsl --list --online  
   Вам отобразится список Ос которые можно установить  
   Пример списка(на момент написание инструкции):

NAME FRIENDLY NAME

Ubuntu Ubuntu

Debian Debian GNU/Linux

kali-linux Kali Linux Rolling

openSUSE-42 openSUSE Leap 42

SLES-12 SUSE Linux Enterprise Server v12

Ubuntu-16.04 Ubuntu 16.04 LTS

Ubuntu-18.04 Ubuntu 18.04 LTS

Ubuntu-20.04 Ubuntu 20.04 LTS

1. Выполните: wsl --install -d Ubuntu-20.04  
   , где Ubuntu-20.04 имя устанавливаемого дистрибутива
2. После установки откроется окно консоли. Вам будет предложено подождать, пока файлы будут распакованы и сохранены на компьютере.  
   По окончанию установки система запросит вас логин и пароль.  
   **Обратите внимание**: при вводе пароля, вводимые символы не отображаются!
3. Все, можно пользоваться!

## Рекомендации

1. Чтобы не было проблем с правами, в Linux’е создайте файл **/etc/wsl.conf**Для создания файла выполните в терминале($): sudo nano /etc/wsl.conf  
   Вставьте содержимое(см. ниже). Ctrl+O - сохранить, Ctrl+X - выйти.

С таким содержанием:

|  |
| --- |
| [automount]  enabled = true  options = "metadata,uid=1000,gid=1000,umask=0022,fmask=11,case=off"  mountFsTab = false  crossDistro = true  [filesystem]  umask = 0022  [network]  generateHosts = true  generateResolvConf = true  [interop]  enabled = true  appendWindowsPath = true |

После чего перезапустите WSL, командой: wsl --shutdown  
И снова запустите, командой: wsl

1. Установите [терминал Windows](https://apps.microsoft.com/store/detail/windows-terminal/9N0DX20HK701?hl=ru-ru&gl=ru)(с доп настройками)

Ссылки на доп.материалы

## Ссылки по Django

* [Официальная документация (eng)](https://docs.djangoproject.com/en/3.0/)
* [Самая полная документация по Django (rus)](https://django.fun/docs/django/ru/3.0/)
* [Хорошая документация, тоже по 1 версии (rus)](https://tutorial.djangogirls.org/ru/)
* [Еще одна хорошая документация](https://metanit.com/python/django/)

Ссылки обновлены: декабрь 2019

## Ссылки по html и css

* [Отличный сайт по html и css](http://htmlbook.ru/html) и [его новая версия](https://html5book.ru/html-html5/)
* [Отличный сайт от mozilla](https://developer.mozilla.org/ru/docs/Web/HTML)

## Using a template to edit models

1. Create a new repo from a [template](https://github.com/SergeyMi37/django-adminlte)

2. Add a directory of Django models [for example locallibrary](https://github.com/SergeyMi37/django-locallibrary-tutorial)

3. Add this directory to core.settings.py in

INSTALLED\_APPS = [

...

#locallibrary

"catalog",

]

4. Add an icon for the side menu to the CatalogConfig class

catalog.apps.py file

class CatalogConfig(AppConfig):

default\_auto\_field = 'django.db.models.BigAutoField'

name = 'catalog'

icon = 'fas fa-book'

5. create a .env file by copying from env.sample

#APPMSW\_PARAM\_NANE=Basic

APPMSW\_LOGO\_TITLE=AppMsW

#apptools img

APPMSW\_LOGO\_IMG=data:image/png;base64,iVB...

APPMSW\_LOGO\_FOOTER=FooterAppMsW

6. To deploy in Docker on nginx, you need to add a variable to .env

CSRF\_TRUSTED\_ORIGINS=http://192.168.1.66:5085

Как добавить в проект возможность разворачивания в докере

Возьмем проект <https://github.com/mdn/django-locallibrary-tutorial>

И добавим в него из проекта <https://github.com/SergeyMi37/django-adminlte>

Файлы docker-compose.yml, Dockerfile, gunicorn-cfg.py и директорию nginx

В файле settings.py изменим

# Set hosts to allow any app on Railway and the local testing URL

ALLOWED\_HOSTS = ['\*']

# Set CSRF trusted origins to allow any app on Railway and the local testing URL

CSRF\_TRUSTED\_ORIGINS = ['https://\*.railway.app','https://\*.127.0.0.1','http://192.168.0.106:5086']

Добавим разрешения для адресов, хотя это можно автоматизировать как в проекте

<https://github.com/SergeyMi37/django-adminlte>

где переменная CSRF\_TRUSTED\_ORIGINS добавляется из файла .env

в файле docker-compose.yml изменим имя сервисов и портов 5086

а в Dockerfile изменим вызов locallibrary.wsgi

CMD ["gunicorn", "--config", "gunicorn-cfg.py", "locallibrary.wsgi"]

Имя файла в директории nginx заменить на имя сервиса locallib

Как добавить в шаблон модель

1. Создайте новый репозиторий из [шаблона] (https://github.com/SergeyMi37/django-adminlte).

2. Добавьте каталог моделей Django [например, locallibrary](https://github.com/SergeyMi37/django-locallibrary-tutorial)

3. Добавьте этот каталог в core.settings.py в

INSTALLED\_APPS = [

...

#locallibrary

"каталог",

]

4. Добавьте в класс CatalogConfig иконку бокового меню.

Файл каталога.apps.py

класс CatalogConfig (AppConfig):

default\_auto\_field = 'django.db.models.BigAutoField'

имя = 'каталог'

icon = 'фас-книга'

5. создайте файл .env, скопировав из env.sample.

#APPMSW\_PARAM\_NANE=Базовый

APPMSW\_LOGO\_TITLE=AppMsW

#apptools изображение

APPMSW\_LOGO\_IMG=данные:изображение/png;base64,iVB...

APPMSW\_LOGO\_FOOTER=FooterAppMsW

6. Для деплоя в Docker на nginx нужно добавить переменную в .env

CSRF\_TRUSTED\_ORIGINS=http://192.168.1.66:5085

Как добавить и настроить мультиязычность.

Первый раз сделал в проекте <https://github.com/SergeyMi37/apptools-django>

Эта инструкция как добавлял в шаблон <https://github.com/SergeyMi37/django-adminlte>

1. Добавить в settings.py

# Internationalization

# https://docs.djangoproject.com/en/3.0/topics/i18n/

# LANGUAGE\_CODE = 'ru' # язык сайта по умолчанию # https://vivazzi.pro/ru/it/translate-django/

#LANGUAGE\_CODE = 'en-US' # de-DE ru-RU

LANGUAGE\_CODE = 'en'

TIME\_ZONE = 'UTC'

USE\_I18N = True

USE\_L10N = True

USE\_TZ = True

# https://egorovegor.ru/django-multiple-language-support/

LANGUAGE\_SESSION\_KEY = 'session\_language\_appname'

LANGUAGE\_COOKIE\_NAME = 'cookie\_language\_appname'

LANGUAGES = (

('en', 'English'),

('ru', 'Russian'),

('de', 'Deutsch'),

)

# месторасположение файлов перевода

LOCALE\_PATHS = (

'locale',

# os.path.join(PROJECT\_DIR, 'locale'),

)

И 'django.middleware.locale.LocaleMiddleware'

MIDDLEWARE = [

    "django.middleware.security.SecurityMiddleware",

    "whitenoise.middleware.WhiteNoiseMiddleware",

    "django.contrib.sessions.middleware.SessionMiddleware",

    'django.middleware.locale.LocaleMiddleware',

    "django.middleware.common.CommonMiddleware",

    "django.middleware.csrf.CsrfViewMiddleware",

    "django.contrib.auth.middleware.AuthenticationMiddleware",

    "django.contrib.messages.middleware.MessageMiddleware",

    "django.middleware.clickjacking.XFrameOptionsMiddleware",

]

После django.contrib.sessions.middleware.SessionMiddleware

2.Добавить в шаблон переключатель в виде формы

В начало загрузку библиотеки работы с переменными lang///

{% load i18n %}

…..

<form action="{% url 'set\_language' %}" method="post">{% csrf\_token %}

          <select name="lang" onchange="document.getElementById('appSubmit2').click();">

              {% get\_current\_language as LANGUAGE\_CODE %}

              {% get\_available\_languages as LANGUAGES %}

              {% get\_language\_info\_list for LANGUAGES as languages %}

              {% for language in languages %}

                  <option title="{{ language.name\_local }} " value="{{ language.code }}"{% if language.code == LANGUAGE\_CODE %} selected="selected"{% endif %}>

                      {{ language.code }}

                  </option>

              {% endfor %}

          </select>

          <input id="appSubmit2" type="submit" style="display: none" value="{% trans 'Change' %}" />

      </form>

Нужно определить url 'set\_language'

Во views.py

@login\_required

def set\_language(request):

    lang = request.POST.get('lang', 'en')

    request.session[dtb.settings.LANGUAGE\_SESSION\_KEY] = lang

    response = HttpResponseRedirect(request.META.get('HTTP\_REFERER', '/'))

    response.set\_cookie(dtb.settings.LANGUAGE\_COOKIE\_NAME, lang)

    return response

функция сохранения выбранного языка в сессии и куках

и редирект текущей страницы для перерисовки контента на выбранный язык

И указатель на эту функцию нужно включить в urls.py

urlpatterns += [

    path(r'set-language/', views.set\_language, name='set\_language'),

]

Во все models, forms другие модули добавить

from django.utils.translation import ugettext as \_

или для версии Django >4.\*

from django.utils.translation import gettext as \_

и все текстовые фразы обрамить в функцию \_

\_('Management Portal')

Для шаблонов применяем

{% trans 'login' %}

Для перевода

Которая при выполнении кода

Соберет все фразы и включит в файлы