python manage.py createsuperuser

Кстати, если вы забыли пароль, то его можно скинуть

python manage.py changepassword <user\_name>

python manage.py runserver

python manage.py makemigrations

python manage.py migrate

source venv/bin/activate

venv\scripts\activate

python manage.py shell

python3 -m venv venv

from newapp.models import \*

from authors.models import \*

django-admin startproject NewsPaper

python3 manage.py startapp news

u1 = User.projects.create\_user(username='Ser')

**Базовые управляющие команды**

Управляющие команды *Django* — это команды, которые позволяют выполнять работу с вашим проектом, не меняя при этом его исходного кода, например, команда как-либо изменить базу данных, создать нового пользователя и т. д.

Например, управляющая команда, создающая новый проект, вам уже известна:

django-admin startproject <**имя** проекта>

Команды имеют примерно следующий вид, как не сложно догадаться:

manage.py <**команда**> <**набор** параметров>

Таких команд, на самом деле, довольно много, но сегодня мы рассмотрим только основные. Если перейти в [документацию](https://docs.djangoproject.com/en/3.1/ref/django-admin/), можно просмотреть полный список всех команд.

Но здесь есть одна очень важная деталь. Все команды из документации описаны довольно непривычно. Например, очень знакомая нам команда для запуска сервера разработки там описана следующим образом:

django-admin runserver [addrport]

Всё это непривычно. Обычно мы привыкли видеть что-то в виде: manage.py <команда> <параметры>, а в итоге видим следующее: django-admin <команда> <параметры>. Сразу нужно отметить, что разницы никакой здесь нет, команды набранные через *manage.py*или через *django-admin*по своему функционалу и сути ничем не отличаются друг от друга. Однако для того, чтобы использовать команды с помощью *django*-*admin* вам надо перейти в виртуальную среду вашего проекта (*venv/bin/activate — Linux, Macos; venv\scripts\activate — Windows)*и указать в переменную среды DJANGO\_SETTINGS\_MODULE путь к файлу *settings.py*вашего проекта.

Например, это можно сделать так:

* + export DJANGO\_SETTINGS\_MODULE=<путь к файлу настроек>.settings — для *Linux*
  + set DJANGO\_SETTINGS\_MODULE=<путь к файлу настроек>.settings*—*для *Windows*.

После этого вы сможете запускать команды для вашего проекта через *django*-*admin*, т. к. модуль будет знать, к какому именно проекту вы обращаетесь. Тем не менее, если у вас всего один проект, то намного удобнее использовать старый добрый *manage*.*py* файл, как мы и будем делать в дальнейшем.

Здесь нам говорится о том, что мы можем кастомизировать создание нашего приложения.

Например, помимо основного параметра — имя приложения, мы можем ещё указать дополнительные аргументы этой команде. Мы можем создать приложение в другой папке или же распаковать шаблон приложения (об этом чуть позже) прямо в папку с проектом, просто введя вместо пути точку — «.», например:

python manage.py startapp myapp .

Команда создаст приложение прямо в той же папке, что и *manage*.*py*, но можно создать приложение, например, и в другой папке. Или же можно помещать ваши приложения в отдельно заготовленную для этого папку, например, так:

django-admin startapp app apps/app

Одной из фишек базовых команд, про которые вы скорее всего не знали, является возможность создавать проекты или приложения по шаблону.

В данном скринкасте представлен пример создания проекта в *Django* по шаблону, который мы напишем сами.

Создание целых проектов по шаблону может быть очень полезным, если вы взялись, например, за разработку какого-то своего движка на *Django*. В таком случае вы можете сами (или разрешить другим пользователям) создавать проекты по образу и подобию уже написанных вами ранее проектов. Это всё делается очень просто — с помощью шаблонов. При создании проекта помимо пути можно также указать дополнительный аргумент --template и указать шаблон для панели, на основе которого надо будет создать проект.

Но для начала надо подготовить саму болванку, т. е. шаблон. Для этого надо создать новый проект и везде поменять название проекта на {{ project\_name }}.

Например, файл *manage*.*py* выглядел бы так:

#!/usr/bin/env python

"""Django's command-line utility for administrative tasks."""

**import** os

**import** sys

**def** **main**():

"""Run administrative tasks."""

os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', '{{project\_name}}.settings')

**try**:

**from** django.core.management **import** execute\_from\_command\_line

**except** ImportError **as** exc:

**raise** ImportError(

"Couldn't import Django. Are you sure it's installed and "

"available on your PYTHONPATH environment variable? Did you "

"forget to activate a virtual environment?"

) **from** exc

execute\_from\_command\_line(sys.argv)

**if** \_\_name\_\_ == '\_\_main\_\_':

main()

*settings.py*

"""

Django settings for {{project\_name}} project.

Generated by 'django-admin startproject' using Django 3.1.5.

For more information on this file, see

https://docs.djangoproject.com/en/3.1/topics/settings/

For the full list of settings and their values, see

https://docs.djangoproject.com/en/3.1/ref/settings/

"""

**from** pathlib **import** Path

# Build paths inside the project like this: BASE\_DIR / 'subdir'.

BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent

# Quick-start development settings - unsuitable for production

# See https://docs.djangoproject.com/en/3.1/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!

SECRET\_KEY = '{{secret\_key}}' # здесь ещё надо поменять секретный ключ, чтобы в целях безопасности он генерировался всегда по новой

# SECURITY WARNING: don't run with debug turned on in production!

DEBUG = True

ALLOWED\_HOSTS = []

# Application definition

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

MIDDLEWARE = [

'django.middleware.security.SecurityMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',

'django.contrib.messages.middleware.MessageMiddleware',

'django.middleware.clickjacking.XFrameOptionsMiddleware',

]

ROOT\_URLCONF = '{{project\_name}}.urls'

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',

],

},

},

]

WSGI\_APPLICATION = '{{project\_name}}.wsgi.application'

# Database

# https://docs.djangoproject.com/en/3.1/ref/settings/#databases

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': BASE\_DIR / 'db.sqlite3',

}

}

# Password validation

# https://docs.djangoproject.com/en/3.1/ref/settings/#auth-password-validators

AUTH\_PASSWORD\_VALIDATORS = [

{

'NAME': 'django.contrib.auth.password\_validation.UserAttributeSimilarityValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.MinimumLengthValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.CommonPasswordValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.NumericPasswordValidator',

},

]

# Internationalization

# https://docs.djangoproject.com/en/3.1/topics/i18n/

LANGUAGE\_CODE = 'en-us'

TIME\_ZONE = 'UTC'

USE\_I18N = True

USE\_L10N = True

USE\_TZ = True

# Static files (CSS, JavaScript, Images)

# https://docs.djangoproject.com/en/3.1/howto/static-files/

STATIC\_URL = 'https://lms-cdn.skillfactory.ru/static/edx-theme/'

*asgi.py*

"""

ASGI config for project\_name project.

It exposes the ASGI callable as a module-level variable named ``application``.

For more information on this file, see

https://docs.djangoproject.com/en/3.1/howto/deployment/asgi/

"""

**import** os

**from** django.core.asgi **import** get\_asgi\_application

os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', '{{project\_name}}.settings')

application = get\_asgi\_application()

*wsgi.py*

"""

WSGI config for project\_name project.

It exposes the WSGI callable as a module-level variable named ``application``.

For more information on this file, see

https://docs.djangoproject.com/en/3.1/howto/deployment/wsgi/

"""

**import** os

**from** django.core.wsgi **import** get\_wsgi\_application

os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', '{{project\_name}}.settings')

application = get\_wsgi\_application()

После этого можно создавать новый проект по имеющемуся шаблону.

django-admin startproject --**template**=project\_name tutorial

И теперь у вас должно появится приложение по шаблону вашего проекта. Туда уже может быть встроен какой-либо функционал или же, наоборот, можно превратить ваш готовый проект в шаблон, выполнив те же действия.

Можно даже перейти в папку *tutorial* с проектом и попробовать запустить его:

python tutorial/manage.py runserver

Также немаловажной командой при развертывании ваших проектов может быть команда по сбору статических файлов. На ней мы особо останавливаться не будем, т. к. она нужна только в том случае, когда ваш сайт по какой-то причине потерял все статические файлы, такие как стили, js-скрипты и т. д. Если вдруг ваша админ-панель стала похожа на нечто похожее:

Просто введите в корне проекта python manage.py collectstatic. Это должно помочь. Сейчас довольно тяжело найти подходящий пример для этой команды, но поверьте, когда мы будем касаться развертки нашего приложения на сервер, эта команда нам ещё понадобится.

Следующая немаловажная команда — создать суперпользователя.

python manage.py createsuperuser

Далее по инструкции надо будет вписать ему *e-mail,* имя пользователя и пароль (*e-mail* во многих проектах по умолчанию можно оставить пустым). Эта команда тоже нам знакома, она создаёт администратора. Кстати говоря, админов в приложении может быть сколько угодно много.

К слову, если вы вдруг забыли пароль от админа — вовсе не обязательно создавать нового, достаточно сбросить пароль:

python manage.py changepassword <**имя** пользователя>

Главное имя-то вы запомнили?

На этом с базовыми командами всё. Как видите, в них есть как много знакомого, так и довольно много нового. В следующем юните мы научимся работать с командами, относящимися к базе данных.

Помимо основных управляющих команд есть также и команды, специализированные на работе с базой данных.

Многие команды вам уже знакомы. Давайте повторим команды, связанные с миграциями (изменениями уже существующих моделей и добавлением новых моделей в базу данных):

* + python manage.py makemigrations [<имя приложения>]*—*создание миграций (обратите внимание, что имя приложения — параметр необязательный, если его указать, то создаются только миграции для конкретного приложения)
  + python manage.py migrate*—*применить созданные миграции, т.е. внести изменения уже именно в саму базу данных (создать новые или редактировать старые модели).
  + python manage.py showmigrations*—*показать все изменения вносимые в базу данных.

Здесь показан список всех миграций в данном проекте с момента его создания, отсортированный по приложениям. Довольно удобная штука для того, чтобы можно было отслеживать изменения моделей.

* Тем не менее, есть вещи поинтереснее, с которыми вы ещё не сталкивались. Например, можно выгрузить данные из вашей базы данных в каком-либо формате.

Давайте попробуем выгрузить все записи из нашей БД в *JSON-*файл:

python manage.py dumpdata --format=json > mydata.json

В нём обычному человеку мало что удастся понять, т. к. он структурирован специально под задачи *Django*, чтобы эти данные можно было в любой момент загрузить обратно в базу. И перед тем, как мы этим займёмся, давайте попробуем выгрузить наши данные, но уже в другом формате:

python manage.py dumpdata --format=xml > mydata.xml

В итоге получится похожий файл, но уже в *XML-*формате:

Так, ну и теперь попробуем удалить данные через админ-панель и снова загрузить их, но уже через команду loaddata:

python manage.py loaddata mydata.json

В итоге у нас должно появится следующее сообщение в консоли:

Installed 3 **object**(s) from 1 fixture(s)

Это говорит о том, что данные снова загрузились успешно! (Постарайтесь, чтобы в ваших объектах не было полей с русскими буквами, иначе можно жёстко застрять с кодировкой).

Последние две команды могут служить для самых разных целей, но, как правило, в большинстве случаев, они используются для создания тестовых данных. Например, данных для базы данных тестового сервера или же фикстур для тестов (тесты мы затронем немножко попозже, но это тоже довольно интересная вещь). Также не обязательно выгружать целиком данные из всего проекта, можно выгрузить данные из какого-то одного приложения, для этого достаточно добавить аргумент с названием приложения:

python manage.py dumpdata --format=xml sample\_app > sampledata.xml

Параметр --database может указывать на базу данных (названия берутся из настроек), из которой будут выгружаться или в которую будут загружаться данные, на случай если у вас их несколько.

Ну и команда экстерминатус — полная очистка базы данных, т. е. удаление всех данных из таблиц в ней.

python manage.py flush

Если у вас несколько баз данных, то можно указать всё тот же параметр --database и очистить какую-то конкретную БД.

Для того чтобы создать собственную команду надо выбрать приложение, с которым она будет логически связана, и создать в его папке следующую структуру, например так:

sample\_app/management/commands/<имя вашей команды>.py

Далее надо создать класс самой команды. Определенные методы этого класса будут отвечать за то или иное поведение во время выполнения вашей команды.

Пример создания класса-команды:

**from** django.core.management.base **import** BaseCommand, CommandError

**class** **Command**(**BaseCommand**):

help = 'Подсказка вашей команды' # показывает подсказку при вводе "python manage.py <ваша команда> --help"

missing\_args\_message = 'Недостаточно аргументов'

requires\_migrations\_checks = True # напоминать ли о миграциях. Если тру — то будет напоминание о том, что не сделаны все миграции (если такие есть)

**def** **add\_arguments**(self, parser):

# Positional arguments

parser.add\_argument('argument', nargs='+', type=int)

**def** **handle**(self, \*args, \*\*options):

# здесь можете писать любой код, который выполняется при вызове вашей команды

self.stdout.write(str(options['argument']))

Теперь давайте поэкспериментируем с нашей командой. Для начала попробуем её просто запустить. Запускается она также, как и любая другая команда, главное не забудьте, что название вашего файла и есть название самой команды!

python manage.py runmycommand

Для того чтобы создать собственную команду надо выбрать приложение, с которым она будет логически связана, и создать в его папке следующую структуру, например так:

sample\_app/management/commands/<имя вашей команды>.py

Далее надо создать класс самой команды. Определенные методы этого класса будут отвечать за то или иное поведение во время выполнения вашей команды.

Пример создания класса-команды:

**from** django.core.management.base **import** BaseCommand, CommandError

**class** **Command**(**BaseCommand**):

help = 'Подсказка вашей команды' # показывает подсказку при вводе "python manage.py <ваша команда> --help"

missing\_args\_message = 'Недостаточно аргументов'

requires\_migrations\_checks = True # напоминать ли о миграциях. Если тру — то будет напоминание о том, что не сделаны все миграции (если такие есть)

**def** **add\_arguments**(self, parser):

# Positional arguments

parser.add\_argument('argument', nargs='+', type=int)

**def** **handle**(self, \*args, \*\*options):

# здесь можете писать любой код, который выполняется при вызове вашей команды

self.stdout.write(str(options['argument']))

Теперь давайте поэкспериментируем с нашей командой. Для начала попробуем её просто запустить. Запускается она также, как и любая другая команда, главное не забудьте, что название вашего файла и есть название самой команды!rjjh

python manage.py runmycommand

**Проект *News Portal***

Доработайте свой интернет-сервис. Напишите команду для *manage*.*py,* которая будет удалять все новости из какой-либо категории, но только при подтверждении действия в консоли при выполнении команды.

Решение

**from** django.core.management.base **import** BaseCommand, CommandError

**from** sample\_app.models **import** Product, Category

**class** **Command**(**BaseCommand**):

help = 'Подсказка вашей команды'

**def** **add\_arguments**(self, parser):

parser.add\_argument('category', type=str)

**def** **handle**(self, \*args, \*\*options):

answer = input(f'Вы правда хотите удалить все статьи в категории {options["category"]}? yes/no')

**if** answer != 'yes':

self.stdout.write(self.style.ERROR('Отменено'))

**return**

**try**:

category = Category.objects.get(name=options['category'])

Post.objects.filter(category=category).delete()

self.stdout.write(self.style.SUCCESS(f'Succesfully deleted all news from category {category.name}')) # в случае неправильного подтверждения говорим, что в доступе отказано

**except** Post.DoesNotExist:

self.stdout.write(self.style.ERROR(f'Could not find category {}'))