**Конспект**

|  |  |
| --- | --- |
| Автор: | Горюнов Евгений С. |
| Город | Томск |
| mail: | ges1987@list.ru |
| Курс: | Fullstack разработчик на Python (SkillFactory) |
| Год обучения: | 2021.08 – 2022г |
|  |  |

Оглавление

[Библиотеки для Django SkillFactory 2](#_Toc93006325)

[Команды для консоли PyCharm 3](#_Toc93006326)

[Создать виртуальное окружение (имя окруж – venv) 3](#_Toc93006327)

[Активация виртуально окружения 3](#_Toc93006328)

[Просмотр установленных приложений в вирт окруж 3](#_Toc93006329)

[Установка фреймворка Django 3](#_Toc93006330)

[Создание начального проекта Django (имя – project) 3](#_Toc93006331)

[Создание приложения в проекте Django (имя – newapp) 3](#_Toc93006332)

[Переход в папку project 3](#_Toc93006333)

[Запуск сервера PyCharm 3](#_Toc93006334)

[Создание миграций 4](#_Toc93006335)

[Провести миграции 4](#_Toc93006336)

[Создание администратора (первого пользователя) 4](#_Toc93006337)

[Сигналы 5](#_Toc93006338)

[notify\_managers 5](#_Toc93006339)

[Полезное 6](#_Toc93006340)

[Подтверждение email через админку 6](#_Toc93006341)

[Как правильно формировать запросы в БД 6](#_Toc93006342)

[Два интересных способа получения инфы с БД 8](#_Toc93006343)

[return в Django - варианты возвратов 10](#_Toc93006344)

[return redirect 10](#_Toc93006345)

[return render 10](#_Toc93006346)

# Библиотеки для Django SkillFactory

главный фреймворк (модуль Д1)

pip install Django

для авторизации, аутентификации пользователей (модуль Д3)

pip install django-allauth django-allauth

для работы фильтров (модуль Д4)

pip install django-filter django\_filters

для выполнения периодических задач (модуль Д6)

pip install django-apscheduler django-apscheduler

команда для обновления пакетного менеджера pip

pip install --upgrade pip

python -m pip install -U pip второй вариант, если первый вызовет ошибку

# Команды для консоли PyCharm

Все команды для windows среды

## Создать виртуальное окружение (имя окруж – venv)

python -m venv venv

## Активация виртуально окружения

venv\scripts\activate

## Просмотр установленных приложений в вирт окруж

pip list

## Установка фреймворка Django

pip install django

## Создание начального проекта Django (имя – project)

python -m django-admin startproject project

## Создание приложения в проекте Django (имя – newapp)

py manage.py startapp newapp

## Переход в папку project

cd project

переход по конкретному пути

cd project\news

Переход на каталог выше

cd ..

## Запуск сервера PyCharm

py manage.py runserver

http://127.0.0.1:8000/ - порт запуска сервера

для запуска сервера на другом порту, для работы в нескольких проектах одновременно

в конце изменить номер порта

py manage.py runserver 7000

## Создание миграций

python manage.py makemigrations

## Провести миграции

py manage.py migrate

## Создание администратора (первого пользователя)

python manage.py createsuperuser

# Сигналы

## notify\_managers

читать тут более подробно

<https://russianblogs.com/article/4399514519/>

Данный метод нужен для того, чтоб начать выполнять код после какого-либо события, например сохранение в БД информации (в нашем примере), ее изменение и так далее. Сигналов множество, на любой случай

Импортируем модули

def notify\_managers\_ appointment (sender, instance, created, \*\*kwargs):

subject = f'{instance.client\_name} {instance.date.strftime("%d %m %Y")}'

print("МЕТОД НУЖНЫЙ СРАБОТАЛ!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!")

mail\_managers(

subject=subject,

message=instance.message,

)

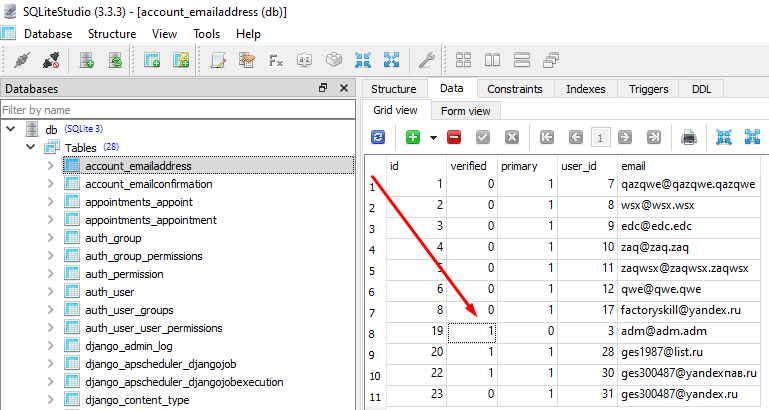
# коннектим наш сигнал к функции обработчику и указываем, к какой именно модели после сохранения привязать функцию

post\_save.connect(notify\_managers\_appointment, sender=Appointment)

# Полезное

## Подтверждение email через админку

Нужно поставить 1 в поле verified таблицы account\_emailaddress



Для чего это нужно, так как при подключении верификации пользователей через почту на сайт не зайдешь, пока почта не будет подтверждена, а почта, как видно из скрина, почти у всех недействительная, тупо для экспериментов служит. Поэтому подтверждаем ее таким способом.

## Как правильно формировать запросы в БД

Если нужно получить один объект (заглавные, либо строчные буквы важно учитывать)

------1---------2------3-----4------5--

Category.objects.get(name='IT')

1 – Category, это наша модель (название таблицы в БД (база данных), он же файл models.py нашего приложения). Если рассматривать с точки зрения кода python– то это обычный класс

2 – objects, это обязательный атрибут, дословно переводится – объекты, то есть команда, которая означает, что нам нужны будут объекты с данной модели

3 – get, значит получить, то есть получить объект с модели Category

4 – name, имя поля, либо имя столбика, колонки в нашей таблице. Если рассматривать с точки зрения кода python – это поле класса.

5 - 'IT', имя объекта в колонке с название «name» в модели Category

Category.objects.get(name='IT').id

Данный метод get нужен чтобы получать, как правило, уникальное значение какой-либо колонки, например, id или pk, или может имени объекта. Если в конце данного запроса поставить точку, ввести какой-либо уточняющий атрибут, например id, то получим его значение. В данном случае мы получает объект, объект – это строчка из таблицы, строчка – это значение ячеек.

Если нужно получить список объектов из модели

--------------------------1------2--------3---------------------4---------------5-----------

Category.objects.filter(pk=1).values('subscribers', 'subscribers\_\_email')

1 – filter, то есть фильтр в переводе, значит будем получать несколько значений, и значит в итоге будем получать список значений

2 - pk=1, какой конкретно объект нужен (строка из таблицы), в нашем случае с рк = 1, primary key, первичный ключ как таковой.

3 – values, значения, это имена колонок в таблице, либо значение ключей в запросе

4 - 'subscribers', 'subscribers\_\_email' – конкретные имена колонок в таблице Category

5 – (\_\_) – двойное подчеркивание, это знак перехода в соседнюю таблицу, то есть данная колонка не содержит информацию, а лишь id и ссылки на соседнюю таблицу, на нужную колонку в соседней таблице. Двойное подчеркивание символизирует связь, то есть две черточки, типа как две линии.

Механизм таков: имя колонки нашей таблички (его имя) \_\_ (двойное подчёркивание) имя колонки из соседней таблички.

subscribers – данное поле содержит лишь pk или id юзеров

subscribers\_\_email – через поле subscribers проникаем в другую табличку в поле email и получаем их значение

В данном случае, чтоб инфу как-то разграничить, ответ мы получаем не просто набором значений ячеек, не тупое перечисление, а сначала имя колонки, а затем значение

[{‘<имя колонки1>’: <значение1>, ‘<имя колонки2>’:<значение2>}]

Пример:

<QuerySet [{'subscribers': 2, 'subscribers\_\_email': 'rom@rom.rom'}, {'subscribers': 3, 'subscribers\_\_email': 'ges1300487@yandex.ru'}, {'subscribers': 6, 'subscribers\_\_email': 'ges1987@list.ru'}]>

Получили мы объект QuerySet (набор запросов в переводе), данное слово оно является тегом, то есть просто для нас, как пометка, для кода оно не существует (лишь в случаях, когда по этому тегу нужно напрямую обратиться и переопределить его).

Для нас важен список (квадратные скобки []) и его содержимое, а содержимое – это набор словарей. Почему именно словари, чтоб была возможность обращаться (найти) нужное значение. Идея следующая, мы получили на наш запрос наш объект (или объекты), то есть нашу строчку из таблицы, и чтоб в этой строчке суметь взять нужное значение, джанга каждое значение пометил из какого столбика оно взято, то есть создал словарь (фигурные скобки {} ), то есть ключ (key, первое значение) – это пометка, а второе значение – это значение самой ячейки.

Вариант того, как нужно отображать наш объект при обычном запросе, без уточнения (.id например при get запросе, либо values при filter запросе) мы указываем в самой модели таким образом:

def \_\_str\_\_(self):

return f'{self.name}'

то есть вывести в админ панель и про обращении имя объекта, можно любую колонку указать

Формируем запрос из БЗ из модели Category и присваиваем его результаты переменной categorys запрос формируется следующим образом:

\_\_\_\_\_\_\_\_\_\_\_\_1\_\_\_\_\_\_\_\_2\_\_\_3\_\_\_4\_\_\_\_\_\_\_\_5\_\_\_\_\_\_\_\_\_\_\_\_\_\_6\_\_\_\_\_\_\_\_\_\_\_\_\_7\_\_\_\_\_\_\_\_\_\_\_\_\_\_8\_\_\_\_\_\_\_

categorys = Category.objects.all().values('subscribers', 'subscribers\_\_username', 'name', 'subscribers\_\_email')[:1]

1-из модели Category (файл models.py приложения news\_app) 2-взять объекты, значения(objects) 3-все(all()) 4-по полям (либо столбикам, либо колонкам таблицы) по названиям 5, 6, 7, 8

## Два интересных способа получения инфы с БД

Данные списки выводят инфу с базы данных через связь многие-ко-многим, то есть начали с модели Category, далее перешли через промежуточную табличку Category\_ Subscribers и далее попали в модель Юзеров от куда вытащили из поля (столбца) email почту всех Юзеров. Что удивительно странно, что поддерживается конструкция Category.objects.get(pk=pole\_test).subscribers.all(), то есть прикол в том, что get запрос подразумевает только один объект, а мы получаем целый список. И далее обращается к каждому объекту

1 способ – более короткий.

pole\_spisok2 = Category.objects.get(pk=pole\_test).subscribers.all()

print(('pole\_spisok2:', pole\_spisok2)

for qwe in pole\_spisok2:

print(qwe.email)

отображение в шаблоне

{% for categorys in pole\_spisok\_html2 %}

<li> {{ categorys.email }} </li>

{% endfor%}

Вывод в консоль:

pole\_spisok2: <QuerySet [<User: Roman>, <User: admin>, <User: Evgeni>]>

Итоговый результат одинаковый у двух вариантов

rom@rom.rom

ges1300487@yandex.ru

ges1987@list.ru

2 способ – более длинный во всех моментах

pole\_spisok3 = Category.objects.filter(pk=pole\_test).values('subscribers\_\_email')

print(('pole\_spisok3:', pole\_spisok3)

for qwe in pole\_spisok3:

print(qwe.get('subscribers\_\_email',))

отображение в шаблоне

{% for categorys in pole\_spisok\_html3 %}

<li> {{ categorys.subscribers\_\_email }} </li>

{% endfor%}

<hr>

Вывод в консоль

pole\_spisok3: <QuerySet [{'subscribers\_\_email': 'rom@rom.rom'}, {'subscriber

s\_\_email': 'ges1300487@yandex.ru'}, {'subscribers\_\_email': 'ges1987@list.ru'}]>

# return в Django - варианты возвратов

## return redirect

return redirect ('test')

вернет (загрузит) представление, указанное в utls.py после исполнения кода

(path('appoint/', AppointView.as\_view(), name='test'), )

Важная особенность, если мы загружаем представление, то загружается и get запрос и весь код, связанный с ним, чтоб этого не делать, можно вызывать метод render (см ниже)

return redirect('/news/')

перейдет по указанному адресу

## return render

return render(request, 'test.html', {})

Более продвинутый вариант, данный вариант позволяет не вызывать представление целиком, а загружать страничку самостоятельно, например страничку test.html, данные запроса и переменные, которые нам нужны на нашей страничке, которые также можно вычислять в пределах POST функции, очень удобно для отладки приложения, для формирования запросов в БД и получения из нее инфы.

----1-------2---------3------------4-------------------5-----------------6-------

return render(request, 'test.html', {'pole\_test\_html': pole\_test, })

1 – return, вернуть (или загрузить), то есть что в итоге должна нам отдать функция, результат нашей функции

2 – render, это метод, то есть функция должна вызвать данный метод render

3 – request, данные запроса (то есть вся инфа, поступившая с POST формы, с нашей странички), по мимо наших данных в теле (в объекте) request содержится куча всего, посмотреть можно так print(dir(request)), и обраться и извлечь инфу из любого атрибута можно такими командами

<имя переменной> = request.POST['category'], либо так, тоже действует (см ниже)

<имя переменной> = request.POST.get('category'), это равноценные команды, но верхняя короче, значит лучше

4 – 'test.html', шаблон (страничка), которую следует загрузит (именно с окончанием .html, а не имя представления в урлах!)

Следующие атрибуты не обязательные, то есть для нормальной работы у нас уже все есть, это данные запроса (request), и шаблон, который нужно грузить пользователю, но если мы хотим еще какие-то данные закинуть на нашу страничку, то создаем словарь (5-6), который может иметь неограниченное количество переменных

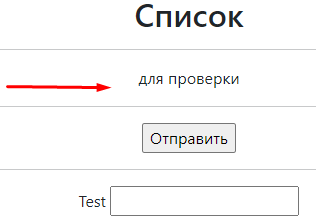
5 и 6 – {'pole\_test\_html': pole\_test, } это обязательно должен быть словарь, то есть ключ (key) и его значение (value). Идея тут следующая, мы на своей страничке размещаем метку (ключ, в нашем случае 'pole\_test\_html', то есть первый аргумент), а после двоеточия – значение метки, то есть переменная в нашем коде в файле представления. Метка на нашей страничке выглядит так {{ pole\_test\_html }}, то есть в любое место в коде воткнули, обязательно в двойных {{ }} фигурных скобках (чтоб html код мог понять, что тут значение переменной из нашего кода джанги), а в джанге в файле представления нашей переменной присваиваем значение, например такое pole\_test = request.POST['test'].

5 - 'pole\_test\_html' – это первый ключ словаря, key

6 - pole\_test – это значение первого ключа словаря, value {key: value, key1: value1, и так далее}

В данном случае, мы вводим в поле Test на нашей страничке любое значение, и тут же выводим его на нашей страничке сверху, либо можно снизу, как удобно

Изображение выглядит как текст

Автоматически созданное описание 

Все что связано с шаблонами менять тут (файл settings.py):

Изображение выглядит как текст

Автоматически созданное описание