ЛАБОРАТОРНАЯ РАБОТА № 8

Связи между таблицами. Методы выбора и обработки записей

Цель: изучить типы связей между таблицами и способы их установки; освоить применение методов выбора и обработки записей из таблиц.

Содержание отчета: титульный лист, цель работы, задание, описание хода выполнения задания со скриншотами и пояснениями, листинги, вывод.

Типы связей между моделями ForeignKey, ManyToManyField, OneToOneField

Рассмотрим, зачем нужны связи между таблицами, и какие типы связей поддерживаются фреймворком Django.

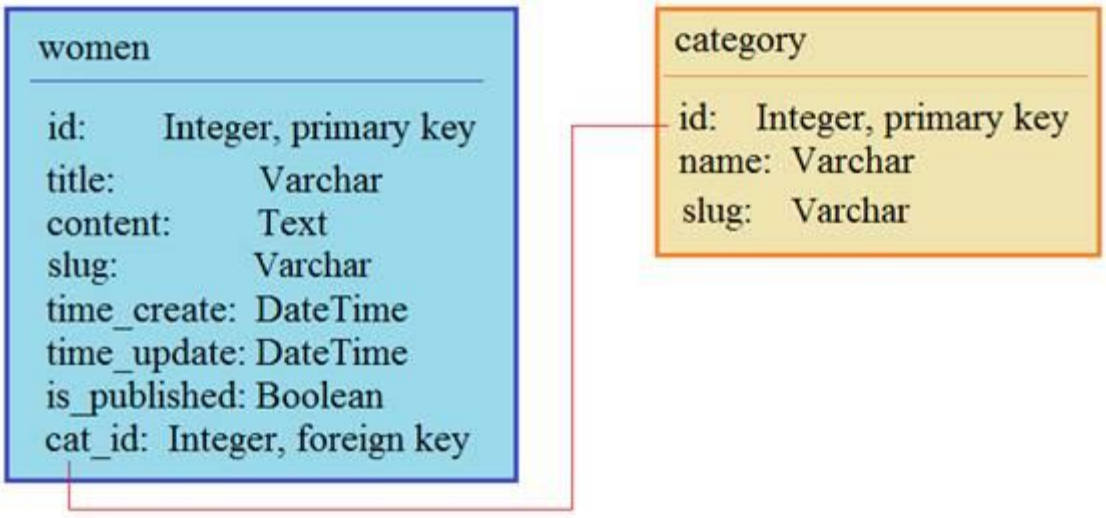
Давайте представим, что для каждой статьи дополнительно нужно хранить категорию, к которой она относится. Если бы мы при этом ограничивались только одной таблицей, то, очевидно, для указания категории статьи нам пришлось бы добавить еще одно поле с названием категории:

id	title	content	time_create	time_update	is_published	slug	category
1	Анд...	биогр...	...	...	1	andj...	Актриса
2	...	...	...	...	1	...	Певица

А теперь представьте, что нам потребовалось изменить название категории, например, вместо «Актриса» записать «Женщины актрисы». Тогда пришлось бы перебирать все записи и категории «Актриса» заменять новым содержимым. Или требуется получить все записи только по актрисам. Тогда снова придется проходить по всей таблице и находить слова «Актриса» в поле **category**. Это будет не самый быстрый процесс. Существует и много других недостатков такого подхода, когда все данные размещаются в одной таблице.

Чтобы ускорить извлечение данных, упростить их редактирование и многое другое, применяется подход разнесения данных по нескольким таблицам и установления связей между ними. Этот процесс получил название **нормализации данных** и составляет отдельный раздел знаний по базам данных.

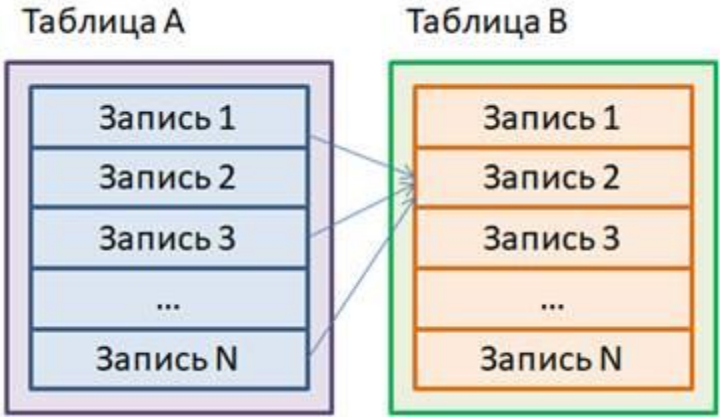
Итак, в нашем проекте, очевидно, нужно определить еще одну таблицу (модель) для категорий и связать ее с таблицей постов:



Для этого мы добавим еще одно поле **cat_id** в таблицу **women**, которое определено как внешний ключ и будет хранить идентификатор категории. А в таблице **category** определим поля: идентификатор **id**, название раздела и слаг. То есть связь таблицы **women** с таблицей **category** выполняется с помощью целочисленного поля **cat_id**. Но какой тип связи здесь будет создан? Это зависит от того, какой класс фреймворка Djnago будет использован для организации связи. Они могут быть следующими:

- **ForeignKey** – для связей Many to One (многие к одному);
- **ManyToManyField** – для связей Many to Many (многие ко многим);
- **OneToOneField** – для связей One to One (один к одному).

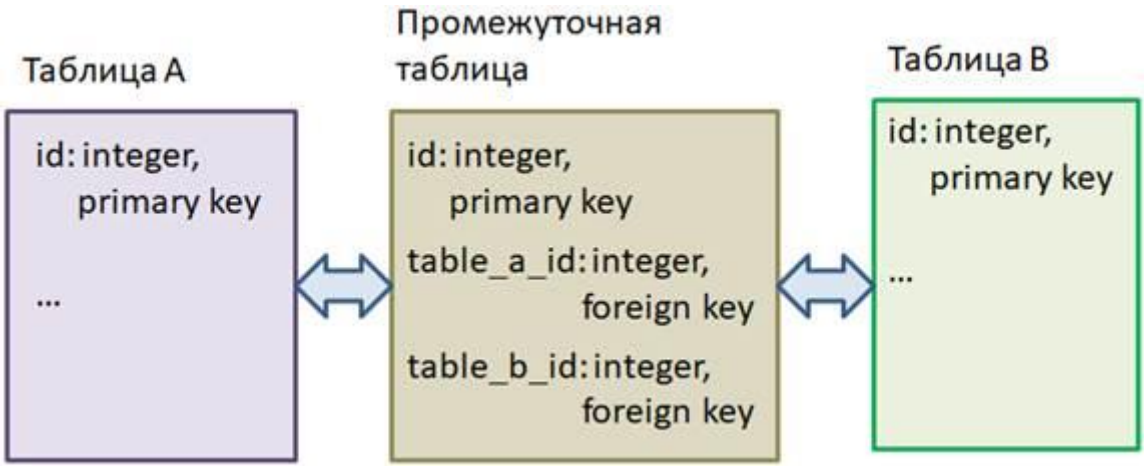
Чаще всего на практике используется отношение многие к одному (**ForeignKey**) и оно подходит для нашего примера, когда несколько постов имеют одну категорию и одна категория может быть связана с множеством постов:



Тип связи Many To One

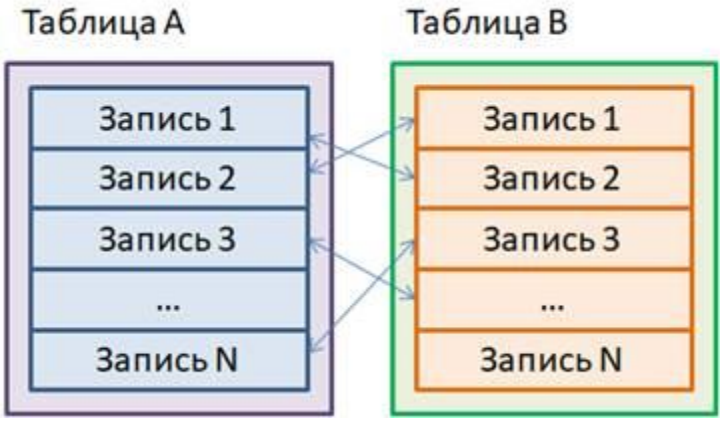
Другой тип связи Many To Many подходит, например, для описания взаимосвязей между студентами и преподавателями. Очевидно, у каждого

студента имеется множество преподавателей и у каждого преподавателя множество студентов. Такой тип связи реализуется через вспомогательную промежуточную таблицу по следующей схеме:



Тип связи Many To Many

Наконец, третий тип связи One To One (один к одному) хорошо подходит, например, для описания взаимосвязи между гражданином и его персональными данными. В частности, так можно связать ИНН с конкретным человеком и, очевидно, у каждого лица в РФ имеется только один ИНН и, соответственно, один ИНН может быть связан только с одним гражданином. То же самое касается и паспортных данных.



Тип связи One To One

Подробнее о классах разных типов связей можно почитать на следующей странице документации:

https://docs.djangoproject.com/en/4.2/ref/models/fields/#module-django.db.models.fields.related

Создание связи many-to-one многие к одному (ForeignKey)

Воспользуемся классом **ForeignKey**, который необходим для организации связей «многие к одному» для отношений между постами и категориями, чтобы каждый пост был связан с одной конкретной категорией, а каждая категория с множеством статей. То есть, здесь слово «многие» относятся к категориям, а слово «один» - к постам. При этом модель **category** называют **первичной**, а модель **women** – **вторичной**.

Класс **ForeignKey** принимает два обязательных аргумента:

- **to** – ссылка или строка класса первичной модели, с которой происходит связывание (в нашем случае это класс **Category** – модели для категорий);
- **on_delete** – определяет тип ограничения при удалении внешней записи (в нашем примере – это удаление из первичной таблицы **Category**).

В свою очередь, опция **on_delete** может принимать следующие значения:

- **models.CASCADE** – удаление всех записей из вторичной модели (например, **Women**), связанных с удаляемой категорией;
- **models.PROTECT** – запрещает удаление записи из первичной модели, если она используется во вторичной (выдает исключение);
- **models.SET_NULL** – при удалении записи первичной модели (Category) устанавливает значение **foreign key** в NULL у соответствующих записей вторичной модели (**Women**);
- **models.SET_DEFAULT** – то же самое, что и **SET_NULL**, только вместо значения NULL устанавливает значение по умолчанию;
- **models.SET()** – то же самое, только устанавливает пользовательское значение;
- **models.DO_NOTHING** – удаление записи в первичной модели (**Category**) не вызывает никаких действий у вторичных моделей.

Подробнее о них можно почитать на странице документации:

https://docs.djangoproject.com/en/4.2/ref/models/fields/#arguments

Посмотрим на конкретном примере, как используется класс **ForeignKey**. Перейдем в файл **women/models.py** и определим еще одну модель для категорий:

```
class Category(models.Model):
    name = models.CharField(max_length=100,
db_index=True)
    slug = models.SlugField(max_length=255,
unique=True, db_index=True)

    def __str__(self):
```

```
        return self.name
```

Здесь все знакомо. Напомним, что параметр **db_index** указывает СУБД индексировать данное поле, чтобы поиск по нему происходил быстрее.

Далее, пропишем дополнительное поле **cat_id** во вторичной модели **Women**:

```
cat = models.ForeignKey('Category',
on_delete=models.PROTECT)
```

(суффикс **_id** для формирования поля **cat_id** Django добавит автоматически).

Обратите внимание, класс первичной модели **Category** указан как строка, потому что модель **Category** в файле **models.py** записана после модели **Women**, поэтому при попытке указать ссылку на этот класс возникнет ошибка. Только поэтому класс указан как строка. Но, вообще, можно записывать и так, и так. Следующий параметр мы определили через функцию **PROTECT** (это ссылка на функцию, а не константа), которая запрещает удаление категорий, используемых во вторичной модели **Women**.

Создание таблиц в базе данных

Формально модели у нас определены, и пришло время создать соответствующие таблицы в БД. Фактически, первую таблицу **women** нужно лишь модифицировать, добавив одно поле, а вторую таблицу создать целиком. Для этого нам надо сформировать новую миграцию командой:

```
python manage.py makemigrations
```

Но полноценно выполниться она не сможет. Дело в том, что в таблице **women** должно добавиться новое поле **cat_id**, ссылающееся на запись первичной таблицы **category**. Но в таблице **women** уже есть записи и при добавлении этого поля оно оказывается для них пустым. СУБД запрещает такую операцию, так как это поле обязательно должно ссылаться на идентификатор записи из связанной таблицы. Отсюда и возникает эта проблема.

Как ее обойти? Временно разрешим записывать в **cat_id** значение NULL. Это можно сделать через параметр **null**:

```
cat = models.ForeignKey('Category',
on_delete=models.PROTECT, null=True)
```

Завершим предыдущую миграцию (выберем 2) и снова запустим команду:

```
python manage.py makemigrations
```

Теперь все прошло успешно, и сформировался еще один файл миграции под номером 0005. Выполним миграции, внесем изменения непосредственно в БД с помощью уже известной нам команды:

python manage.py migrate

Видим, что ошибок никаких нет. Открыв программу SQLiteStudio, увидим две таблицы: **women_category** и **women_women**. Причем в таблице **women_women** появилось новое поле **cat_id** со значениями NULL.

Таблицы сформированы, теперь добавим категории в новую таблицу. Перейдем в терминал, выполним команду:

python manage.py shell_plus

и создадим две записи в таблице **category** (в оболочке shell_plus модели импортируются автоматически):

```
Category.objects.create(name='Актрисы', slug='aktrisy')
Category.objects.create(name='Певицы', slug='pevicy')
```

Затем, у всех записей таблицы **women** установим поле **cat_id** равным 1 (певицы). Сначала выбираем все записи:

```
w_list = Women.objects.all()
```

и обновляем их:

```
w_list.update(cat_id=1)
```

Теперь можно выйти из консоли, убрать аргумент **null=True** у класса **ForeignKey** и снова создать миграции:

python manage.py makemigrations

В появившемся меню выберем пункт 2, и миграция будет создана. Применим ее:

python manage.py migrate

Теперь внешний ключ **cat_id** у нас не может быть пустым.

Работа параметра models.PROTECT

Давайте убедимся в том, что мы не сможем удалить рубрики из первичной модели **Category**, если они используются во вторичной модели **Women**. Для этого снова перейдем в оболочку:

python manage.py shell_plus

и выберем запись с **id** равным 1, так как именно эта категория сейчас используется в постах:

```
c = Category.objects.get(pk=1)
```

И попробуем ее удалить:

```
c.delete()
```

Получим ошибку **ProtectedError**, как раз из-за значения **models.PROTECT** параметра **on_delete**. А вот категорию с **id=2** удалить можно, в этом случае никаких ошибок не возникнет.

Работа параметра models.CASCADE

Посмотрим на порядок работы параметра **models.CASCADE**. Выйдем из оболочки. Пропишем новое значение для **on_delete**:

```
cat = models.ForeignKey('Category',
on_delete=models.CASCADE)
```

Снова перейдем в оболочку и выполним команды:

```
c = Category.objects.get(pk=1)
c.delete()
```

Никаких ошибок не произошло, и при удалении первой категории из таблицы **Category** были удалены также все записи из таблицы **Women**, связанные с этой первой категорией (то есть все записи).

ORM-команды для связи many-to-one

Мы создали новую таблицу для категорий и определили связь многие к одному между таблицей **women** и таблицей **category**. Посмотрим, как через механизм ORM можно работать с этой связью. Перейдем в консоль фреймворка Django:

python manage.py shell_plus --print-sql

и прочитаем запись с **id=1** из таблицы women:

```
w = Women.objects.get(pk=1)
```

В итоге получаем объект класса **Women** с набором локальных атрибутов в виде значений полей:

```
w.title  # 'Анджелина Джоли'
```

```
w.time_create # datetime.datetime(2023, 6, 15)
```

И так далее. Что будет представлять собой атрибут **cat**, который мы в модели объявляли как внешний ключ? Если обратиться к атрибуту **cat_id**:

```
w.cat_id  # 1
```

то увидим значение поля (внешнего ключа) **cat_id**. Но если обратиться к атрибуту:

```
w.cat
```

то в этот момент у нас выполнится еще один SQL-запрос для формирования объекта класса **Category**. И в этом объекте присутствуют соответствующие значения записи из таблицы **category**, связанные с полем **cat_id** из таблицы **women**. В частности, поле name:

```
w.cat.name
```

соответствует строке «Актрисы». Вот так вторичная модель **Women** связывается с первичной моделью **Category** и получает соответствующие данные.

Но можно сделать и наоборот, используя первичную модель **Category**, получить все связанные с ней посты из вторичной модели **Women**. Для этого у любой первичной модели по умолчанию автоматически создается специальное свойство (объект) с именем:

<вторичная модель>_set

И уже с его помощью можно выбирать связанные записи. Давайте сделаем это. Сначала прочитаем запись из таблицы **category**, например, с **id=1**:

```
c = Category.objects.get(pk=1)
```

чтобы получить ссылку на объект записи первичной модели. А затем, используя механизм обратного связывания, прочитаем все связанные с данной категорией посты:

```
c.women_set.all()
```

Если мы хотим переименовать атрибут **women_set**, то для этого в классе **ForeignKey** вторичной модели **Women** следует дополнительно прописать параметр:

```
related_name='posts'
```

Здесь **posts** – это новое имя атрибута для обратного связывания. Чтобы изменения вступили в силу, нужно выйти из оболочки Django, снова зайти, прочитать категорию и отобразить связанные посты с уже новым именем атрибута:

```
c = Category.objects.get(pk=1)
c.posts.all()
```

Вместо метода **all()** мы можем использовать и другие уже известные нам методы, например, **filter()**:

```
c.posts.filter(is_published=1)
```

Фильтрация по внешнему ключу

После того, как мы определили поле **cat** в модели **Women**, его можно использовать в качестве критерия для отбора записей. Например, так:

```
Women.objects.filter(cat_id=1)
```

Получим список всех статей, у которых внешний ключ равен единице. Также мы можем использовать различные фильтры (**lookups**), например, **in** для отбора записей с указанными идентификаторами категорий. Причем записать его можно в двух видах:

```
Women.objects.filter(cat__in=[1, 2])
Women.objects.filter(cat_id__in=[1, 2])
```

Результат будет один и тот же. Или, вместо указания конкретных **id** записей категорий, можно вначале прочитать нужные рубрики, например, все:

```
cats = Category.objects.all()
```

а затем подставить их вместо списка:

```
Women.objects.filter(cat__in=cats)
```

По аналогии используются все другие **lookups** фильтры с внешним ключом. Ограничений здесь никаких нет.

Выборка записей по полям связанных моделей

Помимо фильтров (**field lookups**) мы можем указывать названия полей в связанных таблицах. Для этого параметр следует формировать по правилу:

<первичная модель>__<название поля первичной модели>

Например, так можно выбрать все записи из модели **Women** для определенной категории, используя слаг:

```
Women.objects.filter(cat__slug='aktrisy')
```

Здесь мы обращаемся к первичной модели через атрибут **cat**, который прописан во вторичной модели **Women**, а затем, через два подчеркивания указываем нужное имя поля первичной модели, по которому отбираются записи уже вторичной модели. На выходе получаем список постов для актрис.

Этот синтаксис несколько похож на использование фильтра:

```
Women.objects.filter(cat_id=1)
```

Только здесь вместо указания списка идентификаторов рубрик, используется слаг с определенным названием.

Или можно взять другое поле **name** и по нему производить выборку записей из вторичной модели:

```
Women.objects.filter(cat__name='Певицы')
```

После имени поля можно дополнительно указывать различные фильтры. Например, выберем записи, у которых имя категории содержит букву ‘ы’:

```
Women.objects.filter(cat__name__contains='ы')
```

Если немного изменить этот фильтр:

```
Women.objects.filter(cat__name__contains='цы')
```

то получим записи только по певицам. Или, наоборот, выбрать все категории, которые связаны с записями вторичной модели **Women**, содержащие в заголовке фрагмент строки «ли»:

```
Category.objects.filter(posts__title__contains='ли')
```

На выходе получим набор из нескольких повторяющихся категорий, каждая из которых соответствует определенной записи из модели Women. Если нужно отобрать только уникальные записи (категории), то дополнительно следует указать метод **distinct()**:

```
Category.objects.filter(posts__title__contains='ли').di
stinct()
```

Отображение постов по рубрикам

Сделаем полноценное отображение статей по рубрикам. Сначала в файле **women/urls.py** поменяем в маршруте **category** конвертор с **int** на **slug** и обозначим параметр как **cat_slug**:

```
path('category/<slug:cat_slug>/', views.show_category,
name='category'),
```

После этого перейдем в файл **women/views.py** и отредактируем функцию представления **show_category()** следующим образом:

```
def show_category(request, cat_slug):
    category = get_object_or_404(Category,
slug=cat_slug)
    posts = Women.published.filter(cat_id=category.pk)
    data = {
        'title': f'Рубрика: {category.name}',
        'menu': menu,
        'posts': posts,
        'cat_selected': category.pk,
    }

    return render(request, 'women/index.html',
context=data)
```

Здесь мы проверяем наличие раздела с указанным слагом, и, если его нет в БД, то генерируется исключение **404 PageNotFound**. Если же рубрика найдена, то с помощью менеджера **published** выбираются все посты, у которых категория имеет указанный слаг. Затем формируется словарь **data** с передаваемыми в шаблон **index.html** данными. Причем в **title** мы будем отображать название рубрики. Список **cats_db** удалим.

Рубрики отображаются с помощью пользовательского шаблонного тега, прописанного в файле **women_tags.py**. Здесь в функции **show_categories()** вместо коллекции **cats_db** будем считывать данные из таблицы **category** и передавать в шаблон **list_categories.html**:

```
@register.inclusion_tag('women/list_categories.html')
def show_categories(cat_selected_id=0):
    cats = Category.objects.all()
    return {"cats": cats, "cat_selected":
cat_selected_id}
```

А в самом шаблоне **list_categories.html** изменим отображение ссылок рубрик следующим образом:

```
{% for cat in cats %}
        {% if cat.id == cat_selected %}
                <li
class="selected">{{cat.name}}</li>
        {% else %}
                <li><a href="{{ cat.get_absolute_url
}}">{{cat.name}}</a></li>
        {% endif %}
{% endfor %}
```

Соответственно, метод **get_absolute_url()** необходимо добавить в модель **Category**:

```
class Category(models.Model):
    name = models.CharField(max_length=100,
db_index=True)
    slug = models.SlugField(max_length=255,
unique=True, db_index=True)

    def get_absolute_url(self):
        return reverse('category', kwargs={'cat_slug':
self.slug})

    def __str__(self):
        return self.name
```

Мы сделали отображение статей по категориям, причем URL-адреса будут использовать слаги. После запуска тестового веб-сервера увидим две категории и отображение списка статей строго по каждой из них.

Добавим еще у каждой статьи вывод названия категории и время ее последнего изменения. В шаблоне **index.html** перед заголовком пропишем строчки:

```
<li><div class="article-panel">
        <p class="first">Категория: {{p.cat}}</p>
        <p class="last">Дата: {{p.time_update|date:"d-
m-Y H:i:s"}}</p>
</div>
```

Обращаясь к атрибуту **cat** (а не **cat_id**), мы получаем его строковое представление, то, которое определили в модели **Category** через магический метод **__str__**. То есть **cat** – это объект класса **Category,** и, как вариант, мы можем отображать название категории и через его атрибут **name**:

```
<p class="first">Категория: {{p.cat.name}}</p>
```

Для формирования нужного формата времени используем фильтр **date** с параметрами: день, месяц, год, часы, минуты, секунды. Теперь перед каждой статьей отображается ее категория и время последнего редактирования.

Добавляем связь many-to-many (многие ко многим)

Познакомимся с еще одним типом связей **Many To Many** (многие ко многим). На данный момент мы уже знаем, что она позволяет описывать взаимосвязи между множеством данных двух разных таблиц. Очень частый пример - это реализация информационных тегов для статей. Например, на нашем сайте мы можем определить теги:

оскар; высокие; блондинки; брюнетки; олимпийская чемпионка

И так далее. Понятно, что с одной женщиной можно связать сразу несколько разных тегов и, наоборот, с отдельными тегами – несколько разных постов. Получается взаимосвязь многие ко многим.

Теггирование в Django уже реализовано в модуле **django-taggit**. И с подробной документацией можно ознакомиться на странице:

https://django-taggit.readthedocs.io/en/latest/

В качестве примера самостоятельно добавим функционал теггирования на наш сайт. Сначала в файле **women/models.py** мы определим новую модель для списка тегов следующим образом:

```
class TagPost(models.Model):
    tag = models.CharField(max_length=100,
db_index=True)
    slug = models.SlugField(max_length=255,
unique=True, db_index=True)

    def __str__(self):
        return self.tag
```

А в модели **Women** создадим новое поле с помощью класса **ManyToManyField**:

```
tags = models.ManyToManyField('TagPost', blank=True,
related_name='tags')
```

Здесь первым аргументом мы должны передать или ссылку на класс модели, или имя этого класса в виде строки. Так как класс **TagPost** объявлен после класса **Women**, то используем строку. Параметр **on_delete** здесь не указывается, для поля **ManyToManyField** его прописывать не нужно.

https://docs.djangoproject.com/en/4.2/ref/models/fields/#manytomanyfield

После определения новой модели и изменения существующей нам нужно обновить их структуру в БД. Выполним уже знакомую команду:

python manage.py makemigrations

и применим новые миграции:

python manage.py migrate

Откроем программу SQLiteStudio и увидим, что в итоге были добавлены две новых таблицы. Одна вполне ожидаемая **women_tagpost**, а вторая – вспомогательная с именем **women_women_tags**. Эта вторая таблица будет хранить связи между множеством записей таблицы **women** и **tagpost**. Именно так реализуются связи многие ко многим на уровне БД. При этом в самой таблице **women** не было добавлено ни одного нового поля.

Работа со связью many-to-many через ORM Django

После того, как связь описана и сформированы новые таблицы, рассмотрим, как с ними можно работать с помощью ORM фреймворка Django. Откроем консоль командой:

python manage.py shell_plus

и добавим новые записи в таблицу **tagpost**:

```
TagPost.objects.create(tag='Блондинки', slug='blonde')
TagPost.objects.create(tag='Брюнетки', slug='brunetky')
TagPost.objects.create(tag='Оскар', slug='oskar')
TagPost.objects.create(tag='Олимпиада',
slug='olimpiada')
TagPost.objects.create(tag='Высокие', slug='visokie')
TagPost.objects.create(tag='Средние', slug='srednie')
TagPost.objects.create(tag='Низкие', slug='niskie')
```

Затем первой записи из таблицы **women** (Анджелина Джоли) определим теги: брюнетки; высокие; оскар. Для этого сохраним ссылку на объект записи с **id=1**:

```
a = Women.objects.get(pk=1)
```

и сформируем ссылки на объекты нужных нам тегов. Это можно сделать или так:

```
tag_br = TagPost.objects.all()[1]
```

или сразу распаковать коллекцию с нужными тегами:

```
tag_o, tag_v = TagPost.objects.filter(id__in=[3, 5])
```

Теперь у объекта модели **Women** появился новый атрибут **tags**:

```
a.tags
```

Через него можно выполнять добавление новых тегов к посту. В нашем случае удобно воспользоваться методом **set()** для записи сразу нескольких тегов:

```
a.tags.set([tag_br, tag_o, tag_v])
```

Если нужно добавить только один тег, то для этого существует другой метод **add()**:

```
a.tags.add(tag_br)
```

Причем, если тег для нашего поста уже добавлен, то новой записи в промежуточной таблице не будет, то есть дублирование здесь исключается.

Для удаления тега можно воспользоваться методом **remove()**:

```
a.tags.remove(tag_o)
```

В промежуточной таблице теперь всего две записи.

Чтобы получить список всех тегов для текущей записи, можно воспользоваться очевидной командой:

```
a.tags.all()
```

Или, наоборот, по тегу получить все посты, которые с ним связаны:

```
tag_br.tags.all()
```

Здесь мы используем менеджер с именем **tags**, который определили в параметре **related_name** класса **ManyToManyField**. Соответственно, через него можно вызывать все те же самые методы. Например, для тега **tag_br** добавить второй пост:

```
b = Women.objects.get(pk=2)
tag_br.tags.add(b)
```

Теперь команда:

```
tag_br.tags.all()
```

возвратит две записи.

Вот так можно использовать поле **ManyToManyField**. Отметим один важный момент создания новых записей с тегами. Предположим, что мы бы хотели добавить в БД еще одну известную женщину певицу Ариану Гранде. Если это сделать командой:

```
Women.objects.create(title='Ариана Гранде',
slug='ariana-grande', cat_id=2, tags=[tag_br, tag_v])
```

то получим ошибку, что мы не можем назначать теги еще не сформированной записи. То есть той записи, у которой еще нет идентификатора. Для поля **Many To Many** это необходимая информация, так как именно идентификатор используется для связи поста с тегами и сохраняется в промежуточной таблице. Если этого идентификатора нет, то сформировать такую связь не представляется возможным. Отсюда и ошибка.

Правильно сделать так. Сначала создать запись по этой певице:

```
w = Women.objects.create(title='Ариана Гранде',
slug='ariana-grande', cat_id=2)
```

А уже потом назначить ей необходимые теги:

```
w.tags.set([tag_br, tag_v])
```

Добавление тегов на сайт

Добавим теги на наш сайт. Список всех тегов будем выводить в сайдбаре с помощью шаблонного тега. И, кроме того, у каждой статьи также выводить список тегов, которые к ней относятся.

Пропишем новый маршрут для отображения списка статей по выбранному тегу. В файле **women/urls.py** в коллекцию **urlpatterns** добавим строку:

```
urlpatterns = [
    ...
    path('tag/<slug:tag_slug>/',
views.show_tag_postlist, name='tag'),
]
```

И в модели **TagPost** пропишем метод **get_absolute_url()** для формирования URL-адреса:

```
class TagPost(models.Model):
    tag = models.CharField(max_length=100,
db_index=True)
    slug = models.SlugField(max_length=255,
unique=True, db_index=True)

    def get_absolute_url(self):
        return reverse('tag', kwargs={'tag_slug':
self.slug})

    def __str__(self):
        return self.tag
```

Далее, в файле **women/views.py** объявим функцию представления для нового маршрута:

```
def show_tag_postlist(request, tag_slug):
    tag = get_object_or_404(TagPost, slug=tag_slug)
    posts =
tag.tags.filter(is_published=Women.Status.PUBLISHED)
    data = {
        'title': f'Тег: {tag.tag}',
        'menu': menu,
        'posts': posts,
        'cat_selected': None,
    }

    return render(request, 'women/index.html',
context=data)
```

Мы здесь пытаемся получить тег по его слагу. Если это удается, то затем, читаем список опубликованных постов, связанных с этим тегом. Ниже идет стандартное отображение списка с помощью шаблона **index.html**.

Добавим в функцию представления **index()** параметр **cat_selected** со значением 0:

```
def index(request):
    data = {
        'title': 'Главная страница',
        'menu': menu,
        'posts': Women.published.all(),
        'cat_selected': 0,
    }
```

```
    return render(request, 'women/index.html',
context=data)
```

И, следующим шагом, сформируем новый шаблонный тег для отображения списка тегов в сайдбаре. Откроем файл **women/women_tags.py** и зарегистрируем еще одну функцию:

```
@register.inclusion_tag('women/list_tags.html')
def show_all_tags():
    return {"tags": TagPost.objects.all()}
```

Добавим шаблон **list_tags.html** со следующим содержимым:

```
{% if tags %}
    Теги:</p>
    <ul class="tags-list">
        {% for t in tags %}
        <li><a
href="{{t.get_absolute_url}}">{{t.tag}}</a></li>
        {% endfor %}
    </ul>
{% endif %}
```

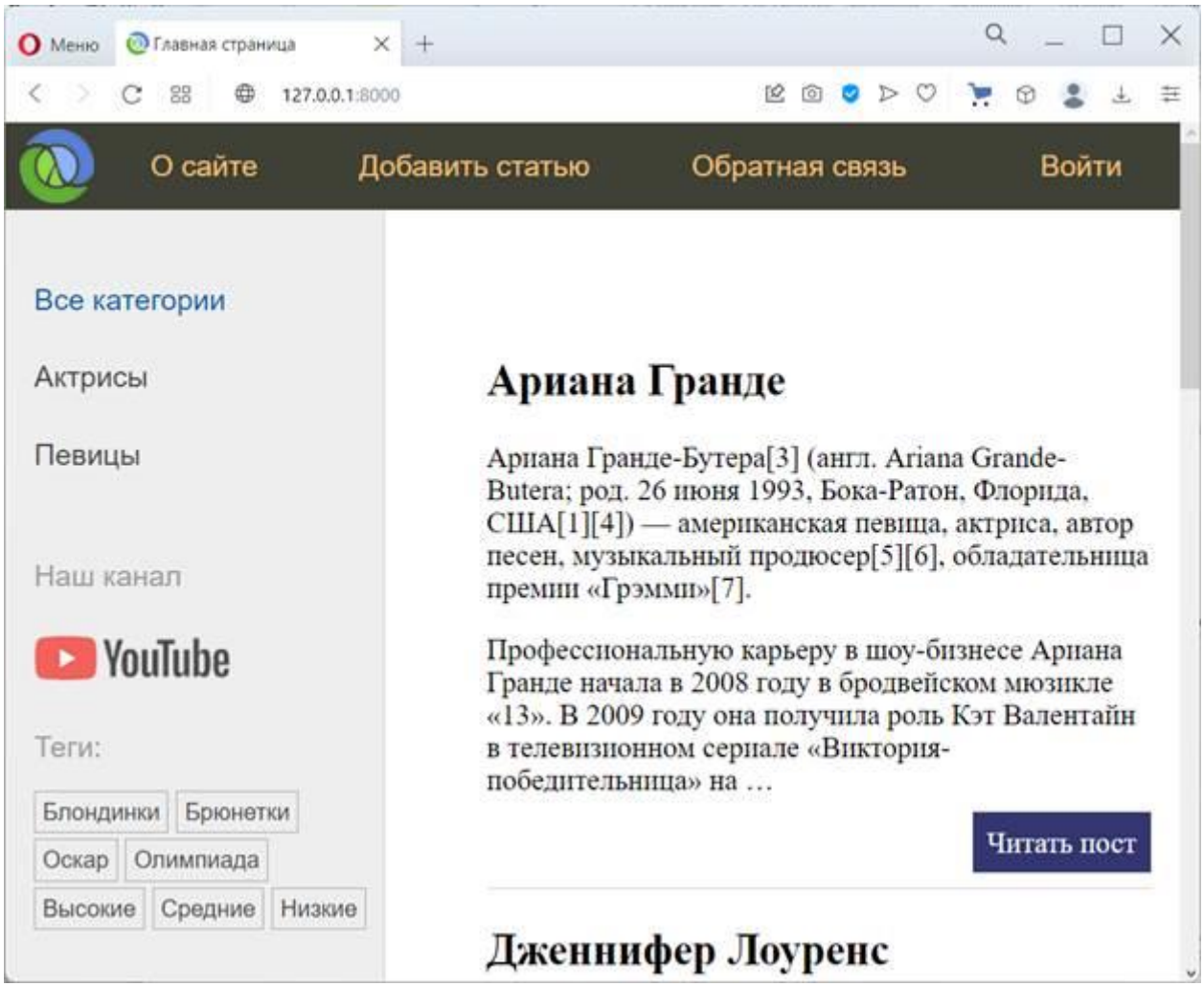
Воспользуемся этим тегом в базовом шаблоне **base.html**:

```
...
<!-- Sidebar слева -->
         <td valign="top" class="left-chapters">
         <ul id="leftchapters">
{% if cat_selected == 0 or cat_selected is None %}
                    <li class="selected">Все
категории</li>
{% else %}
                   <li><a href="{% url 'home' %}">Все
категории</a></li>
{% endif %}
                    {% show_categories cat_selected %}
                    <li class="share">
                    <p >Наш канал</p>
                    <a class="share-yt" href="..."
target="_blank" rel="nofollow"></a>
                    </li>

                    <li>{% show_all_tags %}</li>
         </ul>
</td>
```

```
<!-- Конец Sidebar'a -->
...
```

Запустим тестовый веб-сервер и посмотрим на результат работы сайта:



Также мы можем кликать по тегам и смотреть список статей, связанных с ними. По некоторым тегам не отображается ни одной статьи, но мы пока это оставим в таком виде.

Осталось добавить список тегов, ассоциированных с каждой отдельной статьей. Для этого перейдем в шаблон **women/post.html** и добавим следующие строки:

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

{% block breadcrumbs %}
<!-- Теги -->
{% with post.tags.all as tags %}
{% if tags %}
<ul class="tags-list">
    <li>Теги:</li>
```

```
    {% for t in tags %}
    <li><a
href="{{t.get_absolute_url}}">{{t.tag}}</a></li>
    {% endfor %}
</ul>
{% endif %}
{% endwith %}
{% endblock %}


{% block content %}
<h1>{{post.title}}</h1>

{% if post.photo %}
<p ><img class="img-article-left"
src="{{post.photo.url}}"></p>
{% endif %}

{{post.content|linebreaks}}
{% endblock %}
```

Открываем на сайте страницу с постом и в самом верху видим список тегов, связанных с ней.

Связь one-to-one (один к одному)

Последний тип связи, который мы рассмотрим, это связь **One To One** (один к одному). Ее довольно часто используют для расширения существующих моделей. Например, фреймворк Django изначально предоставляет нам таблицу **user** для хранения информации о пользователях. И ее достаточно для большинства задач. Однако если возникнет необходимость добавить в нее новые поля, то один из вариантов это сделать – создать еще одну таблицу (модель) и присоединить ее к таблице user связью один к одному.

Для демонстрации этого типа связи мы расширим модель **Women**, добавив еще одну модель **Husband** (муж). Будем рассматривать принятую во многих странах ситуацию, когда одна женщина может иметь только одного мужа, а мужчина – одну жену. Здесь подходит связь **One To One**.

Сначала определим модель **Husband** следующим образом:

```
class Husband(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField(null=True)

    def __str__(self):
```

```
        return self.name
```

И добавим в модель **Women** связь один к одному:

```
class Women(models.Model):
    ...
    husband = models.OneToOneField('Husband', on_delete=models.SET_NULL, null=True, blank=True, related_name='wuman')
    ...
```

После этого нам нужно создать файл миграций и применить их:

python manage.py makemigrations

python manage.py migrate

Таблица **husband** создана, наполним ее каким-либо содержимым. Для этого перейдем в консоль фреймворка Django:

python manage.py shell_plus

и выполним следующие знакомые нам команды:

```
h1 = Husband.objects.create(name="Брэд Питт", age=59)
h2 = Husband.objects.create(name="Том Акерли", age=31)
h3 = Husband.objects.create(name="Дэниэл Модер")
```

Теперь мы можем этих мужчин распределить по женщинам. Возьмем первую запись по Анджелине Джоли:

```
w1 = Women.objects.get(pk=1)
```

У этого объекта имеется атрибут **husband**:

```
w1.husband
```

Сейчас он принимает значение NULL. Давайте присвоим ему объект **h1**:

```
w1.husband = h1
```

Теперь Анджелина Джоли снова замужем за Брэдом Питтом. Но в таблице БД этих изменений еще нет. Для этого нужно выполнить метод **save()**:

```
w1.save()
```

Видим, что изменения применились, и в поле **husband** появилось значение 1 – это идентификатор записи из таблицы **husband** для Брэда Питта. Причем мы можем получать данные и через объект w1:

```
w1.husband
```

и через объект h1:

```
h1.wuman
```

Атрибут **wuman** был сформирован благодаря параметру **related_name** класса **OneToOneField**. Если бы мы его не указали, то следовало бы обращаться по имени модели **women** (малыми буквами).

По идее, связь можно было бы установить и с другой стороны (мужа). Предположим, что для второй записи (Марго Робби):

```
w2 = Women.objects.get(pk=2)
```

мы бы хотели указать мужа **h2** (Том Акерли). Сделать это можно и так:

```
h2.wuman = w2
w2.save()
```

Обратите внимание, мы здесь сохраняем именно объект **w2**, так как в нем хранится внешний ключ **husband_id** для связи с таблицей **husband**. То есть, мы должны сохранять ту запись, в которой произошли изменения. Запись **h2** напрямую связь не хранит, а потому и сохранять нечего.

Давайте теперь попробуем Джулии Робертс назначить того же самого мужа **h2** (Тома Акерли):

```
w3 = Women.objects.get(pk=3)
w3.husband = h2
```

Пока никаких ошибок нет. Но при попытке сохранить эту запись:

```
w3.save()
```

получаем ошибку:

IntegrityError: UNIQUE constraint failed: women_women.husband_id

Она, как раз, связана с тем, что запись **h2** уже ассоциирована с записью **w2**, поэтому назначить ее **h2** еще раз другой записи нельзя. В этом смысл связи **One To One** (один к одному).

Мужа **h2** можно было бы назначить женщине **w3**, предварительно удалив связь между **h2** и **w2**. Это можно сделать следующим образом:

```
w2.husband = None
w2.save()
```

Теперь комбинация команд:

```
w3.husband = h2
w3.save()
```

сработают, и у Джулии Робертс появится новый муж. Соответственно, подробную информацию о муже можно посмотреть, обращаясь к объекту **husband**. Например:

```
w1 = Women.objects.get(pk=1)
w1.husband.name
w1.husband.age
```

Мы можем прямо через объект **husband** менять содержимое соответствующей записи:

```
w1.husband.age = 30
w1.husband.save()
```

Теперь Брэд Питт имеет возраст 30 лет.

ORM-команды с классом Q

Рассмотрим более подробно использование ORM Django. Подробную информацию можно посмотреть в документации:

https://docs.djangoproject.com/en/4.2/#the-model-layer

В частности, ссылка на «методы QuerySet»:

https://docs.djangoproject.com/en/4.2/ref/models/querysets/

Перейдем в терминал и запустим консоль:

python manage.py shell_plus --print-sql

Ранее отмечалось, что, если в методе **filter** через запятую указать несколько именованных аргументов, например, так:

```
Women.objects.filter(pk__in=[2,5,7,10],
is_published=True)
```

то сформируется SQL-запрос с условием AND:

WHERE ("women_women"."is_published" AND "women_women"."id" IN (2, 5, 7, 10))

Если в условии нужно использовать логическое **ИЛИ**, а также **НЕ**, то вместо перечисления критериев отбора через запятую, следует использовать специальный класс **Q**. С помощью этого класса можно описывать более сложные критерии (условия), используя специальные операторы:

- & логическое **И** (приоритет 2);
- | логическое **ИЛИ** (приоритет 1 – самый низкий);
- ~ логическое **НЕ** (приоритет 3 – самый высокий).

Посмотрим, как это все работает. Вначале этот класс нужно импортировать:

```
from django.db.models import Q
```

Теперь, если выполнить вот такой запрос:

```
Women.objects.filter(pk__lt=5, cat_id=2)
```

то на выходе получим пустой список, т.к. все записи с id < 5 относятся к первой категории. Но сейчас мы с помощью класса **Q** соединим эти два условия логическим **ИЛИ**:

```
Women.objects.filter(Q(pk__lt=5) | Q(cat_id=2))
```

Теперь видим записи, у которых или **id<5** или **cat_id=2**. Кстати, предыдущий запрос тоже можно записать через класс **Q** следующим образом:

```
Women.objects.filter(Q(pk__lt=5) & Q(cat_id=2))
```

И, наконец, если перед классом прописать тильду, то условие превратится в обратное:

```
Women.objects.filter(~Q(pk__lt=5) | Q(cat_id=2))
```

Здесь мы отбираем записи, у которых **id >=5** или **cat_id=2**.

Разумеется, в методе **filter()** можно комбинировать объекты класса **Q** с обычными аргументами, например, так:

```
Women.objects.filter(Q(pk__in=[1, 2, 5]) | Q(cat_id=2), title__icontains="pa")
```

Тогда первые два выражения будут объединены по ИЛИ, а третье по И:

```
WHERE (("women_women"."id" IN (1, 2, 5) OR "women_women"."cat_id" = 2)
AND "women_women"."title" LIKE '%pa%' ESCAPE '\')
```

Причем первые два условия объединены в круглые скобки и образуют одно единое подусловие. То есть в методе **filter()** каждый аргумент, перечисленный через запятую, образует свое независимое подусловие.

Но мы не можем указывать обычные аргументы до класса **Q**:

```
Women.objects.filter(title__icontains="pa",
Q(pk__in=[1, 2, 5]) | Q(cat_id=2))
```

Получим ошибку. Мы должны такие параметры прописывать либо после класса **Q**, либо обертывать аргумент также в класс **Q,** тогда все отработает без проблем:

```
Women.objects.filter(Q(title__icontains="pa"),
Q(pk__in=[1, 2, 5]) | Q(cat_id=2))
```

Если же нам нужно сделать то же самое, но без дополнительных круглых скобок вокруг **OR**, то следует все прописать через класс **Q** следующим образом:

```
Women.objects.filter(Q(pk__in=[1, 2, 5]) | Q(cat_id=2)
& Q(title__icontains="pa"))
```

Всегда нужно помнить о приоритетах операций: сначала выполняется **НЕ**, затем **И** и, в последнюю очередь, **ИЛИ**.

Методы выбора записей

Рассмотрим некоторые методы извлечения записей. Например, чтобы взять первую запись из выборки, используется метод **first()**:

```
Women.objects.first()
```

Берется первая запись в соответствии с порядком сортировки модели. Мы можем поменять этот порядок и с помощью этого же метода **first()** выбирать разные записи, например, так:

```
Women.objects.order_by("pk").first()
Women.objects.order_by("-pk").first()
```

Или же воспользоваться аналогичным методом **last()** для выбора последней записи из набора:

```
Women.objects.order_by("pk").last()
```

```
Women.objects.filter(pk__gt=5).last()
```

Методы latest и earliest

Если в таблице присутствуют поля с указанием даты и времени, то для таких записей и таких таблиц можно применять методы:

- latest() – выбор записи с самой поздней датой (наибольшей);
- earliest() – выбор записи с самой ранней датой (наименьшей).

Например:

```
Women.objects.earliest("time_update")
Women.objects.latest("time_update")
```

Для чего могут понадобиться такие методы? Например, сделана выборка с сортировкой по какому-либо другому полю (не **time_update**) и из этой выборки нужно получить самую раннюю или самую позднюю запись:

```
Women.objects.order_by('title').earliest("time_update")
```

получаем запись с самой ранней датой изменения.

Методы get_previous_by_, get_next_by_

Если нужно выбрать предыдущую или следующую запись относительно текущей, то в ORM для этого существует два специальных метода, которые выбирают записи по указанному полю с датой и временем. Например, мы выбираем некую запись с **pk=2**:

```
w = Women.objects.get(pk=2)
```

Тогда, для получения предыдущей записи относительно текущей, можно записать:

```
w.get_previous_by_time_update()
```

Здесь суффикс **time_update** – это название поля, по которому определяется предыдущая запись. То есть здесь используется не порядок следования записей в выборке, а временное поле. И уже по нему смотрится предыдущая или следующая запись:

```
w.get_next_by_time_update()
```

Дополнительно в этих методах можно указывать условия выборки следующей или предыдущей записи. Например:

```
w.get_previous_by_time_update(pk__gt=6)
```

выбирается предыдущая запись с **id** больше 6.

Методы exists и count

В ORM Django имеются два полезных метода с высокой скоростью исполнения:

- **exists()** – проверка существования записи;
- **count()** – получение числа записей.

Они часто используются для реализации простых проверок до выполнения других, более сложных запросов.

Добавим в таблицу **Category** еще одну рубрику «Спортсменки»:

```
Category.objects.create(name="Спортсменки",
slug="sportsmenki")
```

Эта рубрика пока пуста, нет ни одной записи с ней связанной. Протестируем метод **exists()**, который возвращает **True**, если записи есть и **False** – в противном случае.

```
c3 = Category.objects.get(pk=3)
c3.posts.exists()
```

Мы обращаемся к атрибуту **posts**, который формируется благодаря наличию параметра **related_name='posts'** в классе **ForeignKey** модели **Women**. Если бы этого параметра не было, то нам следовало бы обращаться к связанным постам так:

```
c3.women_set.exists()
```

Мы видим значение **False** при выполнении запроса, что означает отсутствие каких-либо связанных записей. А, например, для второй категории:

```
c2 = Category.objects.get(pk=2)
c2.posts.exists()
```

получим значение **True**. Вызывая второй метод, можем получить число записей:

```
c2.posts.count()
```

или эту же операцию можно сделать так:

```
Women.objects.filter(cat=c2).count()
```

Методы **exists()** и **count()** можно применить к любой выборке.

Класс F

В предыдущих примерах мы делали выборки, указывая конкретные значения полей, например, так:

```
Women.objects.filter(pk__lte=2)
```

Но что, если вместо 2 нужно прописать значение другого поля таблицы? Просто указать его не получится, например, такая запись:

```
Women.objects.filter(pk__gt="cat_id")
```

приведет к ошибке. Для этого нужно использовать специальный класс **F**, позволяющий нам выполнять подобные операции. Расположен он в ветке django.db.models:

```
from django.db.models import F
```

Строку "cat_id" обернем в класс **F**:

```
Women.objects.filter(pk__gt=F("cat_id"))
```

Получим все записи, кроме первой (с **id=1**). А SQL-запрос будет иметь вид:

SELECT … FROM "women_women" WHERE "women_women"."id" > "women_women"."cat_id"

Здесь условие «id > cat_id» как раз и было сформировано благодаря использованию **F** класса.

Часто подобные операции приходится делать, когда нужно увеличить, например, счетчик просмотра страниц. Давайте для примера в модель **Husband** добавим счетчик числа женитьб для мужчин:

```
class Husband(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField(null=True)
    m_count = models.IntegerField(blank=True,
default=0)

    def __str__(self):
        return self.name
```

Создадим и выполним миграции:

```
python manage.py makemigrations
python manage.py migrate
```

Снова перейдем в консоль фреймворка Django:

python manage.py shell_plus --print-sql

И выполним команду:

```
Husband.objects.update(m_count=F("m_count")+1)
```

Ей соответствует следующий SQL-запрос:

UPDATE "women_husband" SET "m_count" = ("women_husband"."m_count" + 1)

Мы здесь перебираем все записи таблицы **husband** и для каждой из них увеличиваем значение поля **m_count** на единицу. В результате все нули стали единицей. Или можно сделать так. Сначала получить объект записи:

```
h = Husband.objects.get(pk=1)
```

А затем изменить значение поля:

```
h.m_count = F("m_count") + 1
h.save()
```

Теперь счетчик женитьб Брэда Питта стал равен двум.

Метод annotate()

При формировании выборки записей из таблиц БД ORM Django предоставляет возможность формировать дополнительные вычисляемые поля. Что это такое? Предположим, что мы бы хотели автоматически формировать новое булево поле **is_married** для таблицы **husband**, со значениями **True** (истина). Для этого следует воспользоваться специальным методом, который называется **annotate()**, следующим образом:

```
from django.db.models import Value
lst = Husband.objects.all().annotate(is_married=Value(True))
```

Здесь **Value** – это специальный класс, который используется для формирования вычисляемых значений полей таблицы. В данном случае мы просто указали константную величину **True**.

Давайте теперь выведем записи в табличном виде следующим образом:

```
for i, x in enumerate(lst):
    if i == 0:
        print(list(x.__dict__)[1:])
    print(list(x.__dict__.values())[1:])
```

Увидим информацию в таком виде:

['id', 'name', 'age', 'm_count', 'is_married']
[1, 'Брэд Питт', 30, 4, True]
[2, 'Том Акерли', 31, 1, True]
[3, 'Дэниэл Модер', 54, 1, True]

Обратите внимание на последнее поле **is_married**, которого нет в таблице **husband**, но появилось в нашей выборке. Это и есть результат работы метода **annotate()**.

Внутри класса **Value** мы можем записывать любые вычисляемые значения. Например, такие:

```
lst = Husband.objects.all().annotate(is_married=Value(2 + 5))
lst = Husband.objects.all().annotate(is_married=Value("hi "*3))
```

Однако, не можем использовать значения полей. Если записать что то вроде:

```
lst = Husband.objects.all().annotate(is_married=Value("m_count"*3))
```

то получим строку в новом поле **is_married**.

Чтобы оперировать полем в методе **annotate()** следует использовать класс **F**, например, так (у некоторых мужчин **m_count** приравнены нулю):

```
lst = Husband.objects.all().annotate(is_married=F("m_count"))
```

Тогда при выводе увидим:

['id', 'name', 'age', 'm_count', 'is_married']
[1, 'Брэд Питт', 30, 4, 4]
[2, 'Том Акерли', 31, 1, 1]
[3, 'Дэниэл Модер', 54, 0, 0]

Поле **is_married** стало принимать те же значения, что и поле **m_count**. Конечно, в таком действии особого смысла нет. Часто новые поля содержат результаты вычислений на основе предыдущих полей. Например, можно условно вычислить стаж работы, основываясь на возрасте:

```
lst = Husband.objects.all().annotate(work_age=F("age") - 20)
```

Мы здесь условно полагаем, что работать начинают в 20 лет. Получим результаты:

['id', 'name', 'age', 'm_count', 'work_age']
[1, 'Брэд Питт', 30, 4, 10]

[2, 'Том Акерли', 31, 1, 11]
[3, 'Дэниэл Модер', 54, 0, 34]

Или можно сделать так:

```
lst = Husband.objects.all().annotate(work_age=F("age")
- 20, salary=F("age") * 1.10)
```

Сформировалось сразу два новых поля **work_age** и **salary** на основе существующего поля **age**. Наконец, можно оперировать сразу несколькими ранее определенными полями, например:

```
lst = Husband.objects.all().annotate(salary=F("age") *
1.10 - F("m_count") * 5)
```

Агрегирующие функции

Рассмотрим несколько агрегирующих методов. С одним из них мы уже знакомы – это метод **count()**, который подсчитывает число записей. Например, можно определить число записей в таблице **women**:

```
Women.objects.count()
```

В результате у нас формируется SQL-запрос с вызовом функции **COUNT()** на уровне самой СУБД:

SELECT COUNT(*) AS "__count" FROM "women_women"

Число записей подсчитывается в БД, а нам возвращается вычисленный результат. Это большой плюс такого подхода: нам нет необходимости передавать большие блоки данных лишь для того, чтобы, скажем, подсчитать число записей. Все делается «на месте» в БД, и это часто заметно ускоряет подобные процессы. Именно поэтому так важны агрегирующие функции. Информацию о них можно почитать на странице документации:

https://docs.djangoproject.com/en/4.2/ref/models/querysets/#aggregation-functions

ORM Django поддерживает следующие основные команды агрегации: **Count**, **Sum**, **Avg**, **Max**, **Min**. Сначала их следует импортировать из ветки django.db.models:

```
from django.db.models import Count, Sum, Avg, Max, Min
```

И прописывать в специальном методе **aggregate()**. Например, воспользуемся моделью **Husband** и найдем минимальный возраст мужчин:

```
Husband.objects.aggregate(Min("age"))
```

На выходе получим словарь:

{'age__min': 30}

где ключ формируется, как имя поля, два подчеркивания и имя агрегирующей функции, а далее значение этого поля. В данном случае получили минимальный возраст, равный 30.

Также можно прописывать сразу несколько агрегирующих функций через запятую, например:

```
Husband.objects.aggregate(Min("age"), Max("age"))
```

Получим словарь:

{'age__min': 30, 'age__max': 54}

Здесь уже два ключа с соответствующими именами. Если по каким-либо причинам стандартные ключи нам не подходят, и мы бы хотели их поменять, то делается это так:

```
Husband.objects.aggregate(young=Min("age"),
old=Max("age"))
```

на выходе получим:

{'young': 30, 'old': 54}

С агрегирующими значениями можно выполнять различные математические операции, например:

```
Husband.objects.aggregate(res=Sum("age") - Avg("age"))
```

причем параметр **res** в таком случае прописывать строго обязательно, имя ключа автоматически не генерируется.

По аналогии используются все остальные агрегирующие операции:

```
Women.objects.aggregate(Avg("id"))
```

или так:

```
Women.objects.filter(pk__gt=2).aggregate(res=Count("cat
_id"))
```

Здесь агрегация выполняется не для всех записей, а только для тех, у которых **id** больше 2.

Метод values()

Во всех наших примерах при выборке записей автоматически возвращались все поля. Если это была таблица **women**, то получали девять полей от **id** до **husband_id**. Но часто этого не требуется и достаточно ограничиться несколькими из них. Кроме того, такое ограничение положительно сказывается на скорости обращения к БД.

Для указания нужных полей в выборке, используется метод **values()** с указанием названий полей, например, так:

```
Women.objects.values("title", "cat_id").get(pk=1)
```

На выходе имеем запись только с двумя полями. Причем, если мы укажем взять данные из связанной таблицы для имени категории:

```
Women.objects.values("title", "cat__name").get(pk=1)
```

то ORM Django сформирует SQL-запрос с использованием оператора JOIN:

SELECT "women_women"."title", "women_category"."name" FROM "women_women" INNER JOIN "women_category" ON ("women_women"."cat_id" = "women_category"."id") WHERE "women_women"."id" = 1 LIMIT 21

Благодаря такой конструкции одним запросом выбираются все нужные данные. Или даже так:

```
w = Women.objects.values("title", "cat__name")
```

При выполнении этой строчки ни один SQL-запрос выполнен не был, т.к. запросы в ORM Django «ленивые», обращение к БД происходит только в момент получения данных. Но, если вывести список постов:

```
for p in w:
    print(p["title"], p["cat__name"])
```

то увидим, что для этой операции также был сделан всего один запрос. То есть ORM Django достаточно хорошо оптимизирует процесс обращения к БД.

Группировка записей

Часто вызов агрегирующих функций применяется не ко всем записям, а к группам, сформированным по определенному полю. Например, в таблице **Women** можно сгруппировать записи по **cat_id** и получить две независимые группы записей. Затем к каждой группе применить агрегацию и получить искомые значения.

В качестве примера подсчитаем число постов для каждой группы категорий. Для этого запишем такую команду:

```
Women.objects.values("cat_id").annotate(Count("id"))
```

Ее действие графически можно представить так:

Здесь группировка автоматически выполняется по единственному полю, которое мы выбираем из таблицы. При необходимости можем изменить имя параметра **id__count**, например, на **total**, указав его явно в методе **annotate()**:

```
Women.objects.values('cat_id').annotate(total=Count('id'))
```

Мы можем прописывать другие методы после метода **annotate()**. Например, так делается отбор рубрик, у которых число постов больше нуля:

```
lst =
Category.objects.annotate(total=Count("posts")).filter(
total__gt=0)
```

Из SQL-запроса видно, что в этом случае происходит группировка по всем полям таблицы, поэтому ее как бы и нет (хотя бы потому, что поле **id** уникально).

Отобразим результаты в консоли:

```
for i, x in enumerate(lst):
     if i == 0:
         print(list(x.__dict__)[1:])
     print(list(x.__dict__.values())[1:])
```

Получим:

['id', 'name', 'slug', 'total']
[1, 'Актрисы', 'aktrisy', 5]
[2, 'Певицы', 'pevicy', 5]

По аналогии мы можем отобрать все теги из таблицы **tagpost**, которым соответствует хотя бы одна статья:

```
lst =
TagPost.objects.annotate(total=Count("tags")).filter(to
tal__gt=0)
```

Воспользуемся этим запросом для вывода только значащих тегов. Для этого перейдем в файл **women_tags.py** и перепишем функцию **show_all_tags()** следующим образом:

```
@register.inclusion_tag('women/list_tags.html')
def show_all_tags():
    return {"tags":
TagPost.objects.annotate(total=Count("tags")).filter(to
tal__gt=0)}
```

Запускаем тестовый веб-сервер, и у нас отображается только два тега. Но присутствуют три рубрики, одна из которых («Спортсменки») пустая. Сделаем то же самое для вывода рубрик:

```
@register.inclusion_tag('women/list_categories.html')
def show_categories(cat_selected_id=0):
    cats =
Category.objects.annotate(total=Count("posts")).filter(
total__gt=0)
    return {"cats": cats, "cat_selected":
cat_selected_id}
```

Теперь видим только заполненные рубрики.

Вычисления на стороне СУБД

Фреймворк Django содержит набор функций, позволяющих выполнять вычисления, связанные с полями таблицы, на стороне СУБД. Полный их список можно посмотреть по ссылке:

https://docs.djangoproject.com/en/4.2/ref/models/database-functions/

Фактически, здесь приведены обертки над функциями, которые выполняются в СУБД. Этих функций достаточно много. Это и функции работы со строками, датой, математические функции и так далее. Использование этих функций является рекомендуемой практикой, т.к. СУБД оптимизированы для

их выполнения. Конечно, все имеет свои разумные пределы и нужно лишь по необходимости прибегать к этому функционалу.

Для примера рассмотрим использование функции **Length** для вычисления длины строки. Сначала нам нужно ее импортировать:

```
from django.db.models.functions import Length
```

Затем аннотируем новое вычисляемое поле, например, для имен из таблицы **husband**:

```
lst = Husband.objects.annotate(len_name=Length('name'))
```

В результате, наряду со всеми стандартными полями, получим дополнительное поле **len_name**:

['id', 'name', 'age', 'm_count', 'len_name']
[1, 'Брэд Питт', 30, 4, 9]
[2, 'Том Акерли', 31, 1, 10]
[3, 'Дэниэл Модер', 54, 0, 12]

По аналогии используются все остальные подобные функции.

Задание

1. Добавить таблицы в БД, установить связи между таблицами (использовать все виды связей).
2. Добавить на сайт теги. Обеспечить вывод информации на странице с использованием тегов.
3. Использовать в программе описанные методы выбора записей, классы Q, F и Value, вычисляемые поля, агрегирующие функции, группировку записей, вычисления на стороне СУБД.