

Работа с ORM, 2 часть



Александр
Бардин



Александр Бардин

Python-разработчик в Open Solutions



[Александр Бардин](#)



План занятия

1. [Связи многие-ко-многим](#)
2. [Обратная связь](#)
3. [М2М в интерфейсе администратора](#)
4. [Оптимизация запросов к БД](#)
5. [Использование Django Debug Toolbar](#)
6. [Миграции](#)



Связи многие ко многим (m2m)

Вы уже изучили создание и использование связей между моделями (таблицами) один ко многим.

В многих ситуациях такого типа связи может быть недостаточно. Например, пользователи и группы, где один пользователь может состоять в нескольких группах, а в группе может быть несколько пользователей. В таких случаях применяют связь многие ко многим - many to many или m2m.

Определение связи многие к многим в моделях. Вам нужно добавить поле `models.ManyToManyField(Person)`, указав модель, с которой устанавливается связь.

```
from django.db import models

class User(models.Model):
    name = models.CharField(max_length=128)

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(User)
```

Создадим группы:

```
>>> group1 = Group(name='Main')
>>> group1.save()
>>> group2 = Group(name='Moderators')
>>> group2.save()
>>> group3 = Group(name='Administrators')
>>> group3.save()
```

Добавим пользователя и попробуем связать его с группой.

```
>>> user1 = User(name='Username')

>>> group1.members.add(u1)
Traceback (most recent call last):
...
ValueError: Cannot add "<User: User object>": instance is...
```

Получим сообщение об ошибке из-за того, что пользователя нужно было сохранить.



После сохранения

```
>>> user1 = User(name='Username')  
>>> user1.save()  
>>> group1.members.add(u1)
```

или


```
>>> user1 = User.objects.create(name='Username')  
>>> group1.members.add(u1)
```

добавление пользователя в группу происходит успешно.

Обратная связь (related name)

Указав для поля атрибут `related_name`, вы можете задать название поля, через которое можно будет получать связанные объекты.

```
class Group(models.Model):  
    name = models.CharField(max_length=128)  
    members = models.ManyToManyField(User, related_name='groups')
```




Например, задав `related_name='groups'`, можно получить группы конкретного пользователя или добавить пользователя в группу.

```
>>> user1.groups.all() # Получение объектов
```

```
Out[2]: <QuerySet [<Group: Group object>]>
```

```
>>> user1.groups.add(group1) # Добавление пользователя в группу.
```




Если `related_name` не задан, название поля для связи формируется как `related_set`. Где вместо `related` указывается название связанного объекта. Например, примеры предыдущего слайда будут выглядеть так:

```
>>> user1.group_set.all() # Получение объектов
```

```
Out[2]: <QuerySet [<Group: Group object>]>
```

```
>>> user1.group_set.add(group1) # Добавление пользователя в группу.
```



Для m2m связей нет принципиальной разницы в какой модели будет добавлено поле `ManyToManyField`. Если задавать `related_name`, то работа с объектами не будет отличаться.


Создадим модели: `User1` связана с `Group1`, а `User2` с `Group2`.

```
class User1(models.Model):
    name = models.CharField(max_length=128)

class Group1(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(User1, related_name='groups')

class Group2(models.Model):
    name = models.CharField(max_length=128)

class User2(models.Model):
    name = models.CharField(max_length=128)
    groups = models.ManyToManyField(Group2, related_name='members'))
```



Посмотрим созданные по этим запросам таблицы. Как вы видите, таблицы отличаются только названиями.

```
create table mods_group1_members
(
  id integer not null primary key autoincrement,
  group1_id integer not null references mods_group1,
  user1_id integer not null references mods_user1
);
```

```
create table mods_user2_groups
( id integer not null primary key autoincrement,
  user2_id integer not null references mods_user2,
  group2_id integer not null references mods_group2
);
```



Работа с объектами тоже не отличается.

Создадим пользователей и группы.

```
>>> user_1 = User1.objects.create(name='user_1')
>>> user_2 = User2.objects.create(name='user_2')

>>> group_1 = Group1.objects.create(name='group_1')
>>> group_2 = Group2.objects.create(name='group_2')
```



Добавим пользователей в группы так:

```
>>> group_2.members.add(user_2)
>>> group_1.members.add(user_1)
```

или так:

```
>>> user_1.groups.add(group_1)
>>> user_2.groups.add(group_2)
```

Получение связанных объектов:

```
>>> group_1.members.all()
```

```
>>> group_2.members.all()
```

```
>>> user_1.groups.all()
```

```
>>> user_2.groups.all()
```


В некоторых случаях может понадобиться хранить дополнительную информацию о связанных объектах. В таком случае можно при создании связи добавить атрибут, указывающий на модель, описывающую связь между объектами.

```
from django.db import models

class User(models.Model):
    name = models.CharField(max_length=128)

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(User, through='Membership')


class Membership(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```



Добавление пользователей в группу в таком случае выполняется через созданную модель `Membership`.

```
>>> user = User.objects.create(name='user')
>>> group = Group.objects.create(name='group')

>>> memebrship = Membership.objects.create(user=user, group=group, date_joined='2018-10-05')
```



Обычные методы создания отношений при использовании дополнительной таблицы работать не будут.


```
# При использовании дополнительных таблиц, вы получите ошибку.  
>>> group.members.add(user)  
>>> group.members.create(name="User 2")  
>>> user.group_set.add(group)
```

Давайте попробуем спроектировать модели для сервиса составления расписаний учебного заведения. Мы создадим модели для учебных групп (классов), преподавателей и аудиторий.

```
class StudyGroup(models.Model):
    name = models.CharField(max_length=30)

class Teacher(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)

class Auditory(models.Model):
    number = models.IntegerField()
    floor = models.IntegerField()
```



Для начала добавим связи между преподавателем и учебной группой с помощью простой связи m2m.

```
class StudyGroup(models.Model):  
    name = models.CharField(max_length=30)  
    teacher = models.ManyToManyField('Teacher', related_name='study_group')
```

Аудиторию свяжем через дополнительную модель, в которой будем учитывать время начала и завершения занятия, а также день недели в который оно проводится. Назовем этот класс `Schedule`.

```
class StudyGroup(models.Model):
    name = models.CharField(max_length=30)
    teacher = models.ManyToManyField('Teacher', related_name='study_group')
    schedule = models.ManyToManyField()

class Schedule():
    group = models.ForeignKey('StudyGroup', on_delete=models.CASCADE)
    auditory = models.ForeignKey('Auditory', on_delete=models.CASCADE)
    time_start = models.TimeField()
    time_end = models.TimeField()
    day_of_week = models.CharField(max_length=20)
```



M2M в интерфейсе администратора

В интерфейсе администратора возможно создавать и редактировать объекты ваших моделей.

Для связанных объектов редактирование на разных страницах может создать трудности, вызвать ошибки, да и просто может быть не удобно.

Для отображение связанных объектов на одной странице в интерфейсе администратора применяются объекты типа inline.

После добавления inline классов, вы увидите на странице редактирования объекта все объекты связанные с ним. Такие связанные объекты можно редактировать или добавлять новые.

Используем модели из примеров выше:

```
class User(models.Model):
    name = models.CharField(max_length=128)

    def __str__(self):
        return self.name

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(User1, related_name='groups')


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

И добавим их в интерфейс администратора:

```
class MembershipInline(admin.TabularInline):
    model = Group.members.through
    extra = 1

class UserAdmin(admin.ModelAdmin):
    inlines = [
        MembershipInline,
    ]

class GroupAdmin(admin.ModelAdmin):
    inlines = [
        MembershipInline,
    ]
    exclude = ('members',)
```



Как вы видите, в отличие от связей `ForeignKey`, в данном случае атрибут `model` задается с указанием специального объекта менеджера, связывающего модели.

```
model = Group1.members.through
```


Если связь осуществляется через выделенный класс:

```
class User(models.Model):
    name = models.CharField(max_length=128)

    def __str__(self):
        return self.name

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(User, through='Membership')


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



```
class Membership(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)


    def __str__(self):
        return '{0}_{1}'.format(self.user, self.group)
```

Связь моделей в интерфейсе администратора настраивается немного иначе.



Обратите внимание на метод `str`, он вызывается во время применения функции `str()` к объекту. В django это происходит, например, при отображении объекта в интерфейсе администратора. Если метод не определен для вашего класса, то будет использована реализация этого метода родительского объекта. В нашем примере `models.Model`. Такое сообщение может оказаться менее информативно.


```
def __str__(self):  
    return '{0}_{1}'.format(self.user, self.group)
```



```
class MembershipInline(admin.TabularInline):  
    model = Membership  
    extra = 1
```

```
class UserAdmin(admin.ModelAdmin):  
    inlines = (MembershipInline,)
```

```
class GroupAdmin(admin.ModelAdmin):  
    inlines = (MembershipInline,)
```



С помощью параметра `extra = 1` можно задать количество дополнительных пустых строк, отображаемых в интерфейсе.

Также вы можете изменить то, как будут отображаться связанные объекты.

```
class MembershipInline(admin.TabularInline):  
    model = Membership  
    extra = 1
```

```
class MembershipInline(admin.StackedInline):  
    model = Membership  
    extra = 1
```




Оптимизация запросов к БД

Для облегчения работы с базами данных в django используется технология ORM, автоматизирующая и облегчающая составление запросов к БД. В некоторых случаях запросы, сформированные ORM, не оптимальны и создают высокую нагрузку на БД.

Далее мы рассмотрим некоторые способы оптимизации запросов, которые генерирует ORM.


prefetch_related

Получим группы модели `Group1`, рассмотренной выше и выведем имена пользователей в каждой группе.

```
groups = Group1.objects.all() # Будет сделан запрос
# SELECT "mods_group1"."id", "mods_group1"."name" FROM "mods_group1"
for group in groups:
    print('Group: {}, Users:'.format(group.name))
    for user in g.members.all():
        print('- {}'.format(user.name))
```

Не смотря на то, что явно мы определяем один запрос `groups = Group1.objects.all()`, на самом деле запросы будут сделаны на каждой итерации цикла. На первый взгляд в этом нет ничего страшного, но представьте, что пользователей у нас в системе несколько тысяч.

Количество запросов к БД в таком случае будет очень большим.



Для улучшения ситуации мы можем использовать метод запроса `prefetch_related`, указав какие связанные объекты нам понадобятся.

```
groups = Group1.objects.all().prefetch_related('members')
```

При этом все пользователи будут получены из БД, сохранены в кэше и использованы при необходимости.



Вопрос

В чем отличие `prefetch_related` от `select_related`?

`select_related` соединяет одну связанную сущность для данной модели (связь «к одному»), а `prefetch_related` присоединяет много сущностей к данной модели (связь «ко многим»).

Пример:

- у модели телефон 1 производитель:
`Phone.objects.select_related('company')`
- в группе много студентов:
`Group.objects.prefetch_related('members')`

only и defer

Методы `only` и `defer` позволяют выбрать в запросе только необходимые поля. Например, у пользователя может быть определены поля `name`, `age` и `email`. Нам при запросе нужно только имя.

```
# Исключим ненужные поля
Person.objects.defer("age", "email")
# Выберем только нужное
Person.objects.only("name")
```

При изменении и последующем сохранении записей запроса с `only` или `defer`, будут сохраняться в БД только выбранные поля.

Методы `values` и `values_list`

Ещё одним способом получить только нужную часть данных является использование методов `values` и `values_list`. При использовании `values` вы получите `QuerySet`, содержащий для каждой записи словарь с выбранными данными.

```
Group.objects.all().prefetch_related('members').values_list('name',  
'members__name')  
# <QuerySet [('group1', 'user1'), ('group1', 'user2'), ('group1', 'user3')]>
```



Использование Django Debug Toolbar

Часто в процессе разработки возникает необходимость отладки вашего приложения или нужно установить причину медленного выполнения запросов.

Для решения подобных проблем существуют различные решения.

Очень часто подобный функционал имеется в интегрированных средах разработки.

Мы рассмотрим использование Django Debug Toolbar.



Для его установки в ваше окружение можно воспользоваться командой:

```
pip install django-debug-toolbar
```

Настройка Debug toolbar

Для включения debug toolbar:

- нужно в файле настроек settings.py добавить в список приложений debug_toolbar,
- убедиться в том, что в нём присутствует модуль django.contrib.staticfiles,
- настроить параметр STATIC_URL.

```
INSTALLED_APPS = [  
    # ...  
    'django.contrib.staticfiles',  
    # ...  
    'debug_toolbar',  
]  
  
STATIC_URL = '/static/'
```



Там же вы должны изменить список MIDDLEWARE:

```
MIDDLEWARE = [  
    # ...  
    'debug_toolbar.middleware.DebugToolbarMiddleware',  
    # ...  
]
```

После этого нужно разрешить доступ к debug-toolbar указав свой ip адрес в settings.py:

```
# для локального сайта нужно указать адрес '127.0.0.1'  
INTERNAL_IPS = ['127.0.0.1']
```




После этого вам нужно основной файл `urls.py` вашего проекта изменить следующим образом:

```
if settings.DEBUG:
    import debug_toolbar
    urlpatterns = [
        path('__debug__/', include(debug_toolbar.urls)),

        # For django versions before 2.0:
        # url(r'^__debug__/', include(debug_toolbar.urls)),

    ] + urlpatterns
```



Пример настроек взят из документации django debug toolbar. Там вы можете ознакомиться с дополнительными настройками:

<https://django-debug-toolbar.readthedocs.io/en/latest/installation.html>

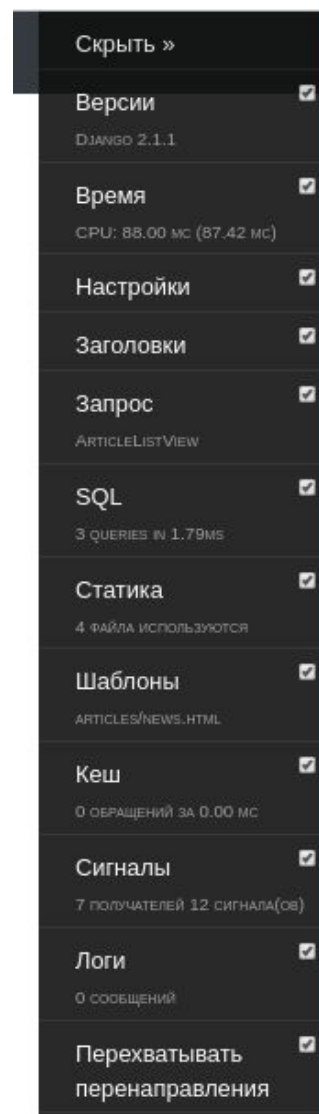
Также обратите внимание, что debug toolbar активен только в режиме отладки вашего приложения (DEBUG = True в settings.py).


Информация, представленная на страницах debug toolbar, относится либо к текущей странице, либо к сайту целиком.

После настройки на вашем справа сайте появится кнопка:



После нажатия на неё вы увидите меню debug toolbar:






На странице «Версии» вы можете посмотреть, какие пакеты установлены в вашем окружении.

Страница «Время» содержит информацию о том, как долго выполнялась обработка страницы на сервере и сколько времени занимала обработка в браузере.


Страница «Настройки» показывает содержимое `settings.py`.



Страница «Заголовки» показывает заголовки запроса, ответа и окружения (сервера).

«Запрос» показывает имеющиеся сессии, куки, параметры, передаваемые в GET или POST запросе, и использованное для обработки представление (view).


«SQL» показывает запросы к БД и места в коде, где эти данные использовались.



На странице «Статика» вы увидите, какие статические файлы (изображения, стили, скрипты, ...) были использованы для отображения страницы и какие доступны.

Страница «Шаблоны» показывает использованные шаблоны и данные, доступные для использования в них.

Страница «Кэш» показывает какие данные были помещены в кэш и использованы для отображения.



Страница «Сигналы» показывает доступные сигналы и их получателей. (В django можно получать информацию об изменении состояния некоторых объектов.

Например, можно отслеживать добавление пользователя в группу или завершение обработки запрашиваемых страниц).


«Логи» показывают события (информация, предупреждения, ошибки) сохраненные системой логирования.



Миграции


В процессе разработки и поддержки проекта часто возникает необходимость изменения модели данных. Каждое такое изменение должно быть применено к базе данных.

Для того чтобы синхронизировать изменения модели данных и структуры таблиц в БД и не менять таблицы вручную, в django имеется механизм миграций.




После создания или изменения модели вы должны выполнить management команду makemigrations.

```
./manage.py makemigrations
```



В команде вы можете указать одно или несколько приложений, для которых вы хотите создать миграции. Иначе будут созданы миграции для всех доступных приложений.

```
./manage.py makemigrations app1
```



Если указать аргумент `empty`, в указанном приложении будет создана пустая миграция, которую вы можете дописать самостоятельно. Например, вы можете реализовать добавление пользователей или наполнение таблиц необходимыми для работы данными.

```
./manage.py makemigrations --empty app1
```

Выполнив такую команду, вы получите модуль с шаблоном миграции:

```
from __future__ import unicode_literals

from django.db import migrations

class Migration(migrations.Migration):

    dependencies = [
        ('mods', '0001_initial'),
    ]

    operations = [

```


Который вы можете отредактировать под собственные нужды:

```
from __future__ import unicode_literals


from django.db import migrations
from django.contrib.auth.models import User

def addsuperuser(apps, schema_editor):
    user = User(pk=1, username="admin", is_active=True,
                is_superuser=True, is_staff=True,
                last_login="2017-09-01T13:20:30+03:00",
                email="email@email.com",
                date_joined="2017-09-01T13:20:30+03:00")
    user.set_password('admin')
    user.save()

...
```





```
...  
class Migration(migrations.Migration):  
  
    dependencies = [  
        ('orders', '0001_initial'),  
    ]  
  
    operations = [  
        migrations.RunPython(addsuperuser),  
    ]
```



Применить миграции вы можете командой `migrate`. В результате будут применены все новые миграции.

```
./manage.py migrate
```



Вы также можете выбрать приложения для миграции, перечислив их после команды:

```
./manage.py migrate app1
```



Просмотреть состояние миграций можно командой:

```
./manage.py showmigrations

...
contenttypes
[X] 0001_initial
[X] 0002_remove_content_type_name
mods
[ ] 0001_initial
[ ] 0002_auto_20181004_1507
```

[] - миграции, которые еще не синхронизированы с БД.



Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задавайте **в чате** мессенджера Slack.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

**Задавайте вопросы и
пишите отзыв о лекции!**

 [Александр Бардин](#)

Александр Бардин