

Лекция 5. Модели. Часть 3.



Работа с объектами модели: создание и получение объектов модели

В файле **views.py** опишем две функции-представления:
index() и create()



```
views x
18 from django.shortcuts import render
19 from django.http import HttpResponseRedirect
20 from .models import Person
21
22
23 # получение данных из бд
24 def index(request):
25     people = Person.objects.all()
26     return render(request, "index.html", {"people": people})
27
28
29 # сохранение данных в бд
30 def create(request):
31     if request.method == "POST":
32         tom = Person()
33         tom.name = request.POST.get("name")
34         tom.age = request.POST.get("age")
35         tom.save()
36     return HttpResponseRedirect("/")
```

В функции **index()** получаем все данные с помощью метода **Person.objects.all()** и передаем их в шаблон **index.html**.

В функции **create()** получаем данные из запроса типа POST, сохраняем данные с помощью метода **save()** и выполняем переадресацию на корень веб-сайта (то есть на функцию **index()**).



```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Модели Django</title>
</head>
<body class="container">
  <form method="POST" action="create/">
    {% csrf_token %}
    <p>
      <label>Введите имя</label><br>
      <input type="text" name="name" />
    </p>
    <p>
      <label>Введите возраст</label><br>
      <input type="number" name="age" />
    </p>
    <input type="submit" value="Сохранить" />
  </form>
  {% if people.count > 0 %}
  <h2>Список пользователей</h2>
  <table>
    <tr><th>Id</th><th>Имя</th><th>Возраст</th></tr>
    {% for person in people %}
    <tr><td>{{ person.id }}</td><td>{{ person.name }}</td><td>{{ person.age }}</td></tr>
    {% endfor %}
  </table>
  {% endif %}
</body>
</html>

```

В папке **templates** определим шаблон **index.html**, который будет выводить данные на веб-страницу.

В начале шаблона определена форма для добавления данных, которые потом будет получать функция create в POST-запросе.

А ниже определена таблица, в которую выводятся данные из переданного из представления набора people.

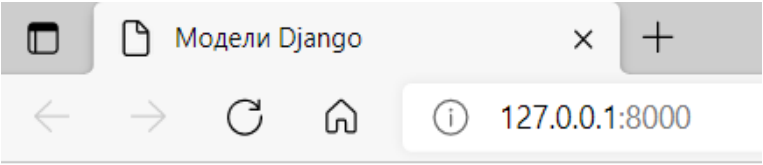


в файле urls.py свяжем маршруты с представлениями:

```
12 including another URLconf
13     1. Import the include() function: from django.urls import include, path
14     2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))
15 """
16 from django.contrib import admin
17 from django.urls import path, re_path
18 from artist import views
19 from django.views.generic import TemplateView
20
21
22 urlpatterns = [
23     path('', views.index),
24     path('create/', views.create),
25     path('about/', TemplateView.as_view(template_name="artist/about.html")),
26     path('contact/', TemplateView.as_view(template_name="artist/contact.html",
27         extra_context={"work": "Проектирование ИС"})),
28     path('products/<int:productid>/', views.products),
29     path('users/', views.users),
30     path('special/', views.special),
31
32 ]
```



Запустим сервер: `python manage.py runserver` и обратимся к приложению в браузере:



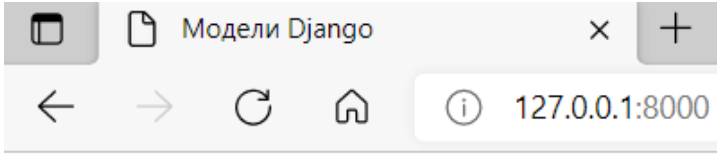
Введите имя

Введите возраст

Сохранить

Список пользователей

Id	Имя	Возраст
1	Иванов Иван	18
2	Петров Петр	23
3	Сидоров Сидор	19



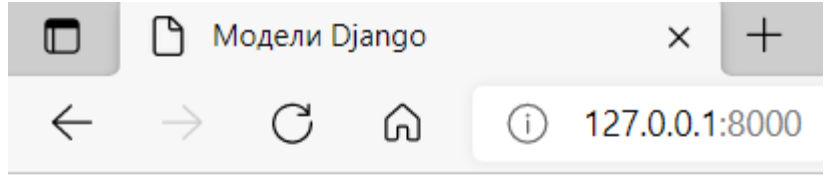
Введите имя

Введите возраст

Сохранить

Список пользователей

Id	Имя	Возраст
1	Иванов Иван	18
2	Петров Петр	23
3	Сидоров Сидор	19
4	Орлов Владимир	35



Введите имя

Введите возраст

Сохранить

Список пользователей

Id	Имя	Возраст
1	Иванов Иван	18
2	Петров Петр	23
3	Сидоров Сидор	19
4	Орлов Владимир	35
5	Соколов Игорь	17

Вначале добавим несколько объектов через форму на веб-странице. После каждого ввода и добавления мы увидим, как на веб-странице в таблице появляются новые данные:



Посмотрим таблицу в SQLiteStudio:

SQLiteStudio (3.3.3)

Database Structure View Tools Help

artist_person (db1)

Структура Данные Ограничения Индексы Триггеры DDL

Табличный вид Форма

Отфильтровать да...

	id	name	age
1	1	Иванов Иван	18
2	2	Петров Петр	23
3	3	Сидоров Сидор	19
4	4	Орлов Владимир	35
5	5	Соколов Игорь	17



Работа с объектами модели: редактирование и удаление объектов модели

```
def edit(request, id):
    try:
        person = Person.objects.get(id=id)

        if request.method == "POST":
            person.name = request.POST.get("name")
            person.age = request.POST.get("age")
            person.save()
            return HttpResponseRedirect("/")
        else:
            return render(request, "artist/edit.html", {"person": person})
    except Person.DoesNotExist:
        return HttpResponseRedirect("<h2>Person not found</h2>")
```

Добавим в файл представлений **views.py** функции **edit()** и **delete()**, которые будут выполнять редактирование и удаление записей из таблицы:

```
# удаление данных из бд
def delete(request, id):
    try:
        person = Person.objects.get(id=id)
        person.delete()
        return HttpResponseRedirect("/")
    except Person.DoesNotExist:
        return HttpResponseRedirect("<h2>Person not found</h2>")
```




```

views x
# изменение данных в бд
def edit(request, id):
    try:
        person = Person.objects.get(id=id)

        if request.method == "POST":
            person.name = request.POST.get("name")
            person.age = request.POST.get("age")
            person.save()
            return HttpResponseRedirect("/")
        else:
            return render(request, "artist/edit.html", {"person": person})
    except Person.DoesNotExist:
        return HttpResponseRedirect("/")

# удаление данных из бд
def delete(request, id):
    try:
        person = Person.objects.get(id=id)
        person.delete()
        return HttpResponseRedirect("/")
    except Person.DoesNotExist:
        return HttpResponseRedirect("/")

```

Функция **edit()** выполняет редактирование объекта. Функция в качестве параметра принимает идентификатор объекта в базе данных и по этому идентификатору ищется объект с помощью метода **Person.objects.get(id=id)**.

Исключение **Person.DoesNotExist** – для случая, если будет передан несуществующий идентификатор. Если объект не будет найден, то пользователю возвращается ошибка 404 через вызов **return HttpResponseRedirect()**.

Если объект найден, то обработка делится на две ветви. Если запрос **POST**, то есть если пользователь отправил новые измененные данные для объекта, то эти данные сохраняются в базе и идет переадресация на корень веб-сайта. Если запрос **GET**, то пользователю отображается страница **edit.html** с формой для редактирования объекта.

Функция **delete** аналогичным образом находит объект и выполняет его удаление.



Создадим в папке **templates** файл **edit.html** с определением формы для редактирования объекта:

```
edit x
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8" />
5      <title>Модели в Django</title>
6  </head>
7  <body class="container">
8      <form method="POST">
9          {% csrf_token %}
10         <p>
11             <label>Введите имя</label><br>
12             <input type="text" name="name" value="{{person.name}}" />
13         </p>
14         <p>
15             <label>Введите возраст</label><br>
16             <input type="number" name="age" value="{{person.age}}" />
17         </p>
18         <input type="submit" value="Сохранить" >
19     </form>
20 </body>
21 </html>
22
```

По нажатию на кнопку **Сохранить** введенные в форму данные будут уходить по тому же адресу в запросе POST.



```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Модели в Django</title>
</head>
<body class="container">
  <form method="POST" action="create/">
    {% csrf_token %}
    <p>
      <label>Введите имя</label><br>
      <input type="text" name="name" />
    </p>
    <p>
      <label>Введите возраст</label><br>
      <input type="number" name="age" />
    </p>
    <input type="submit" value="Сохранить" />
  </form>
  {% if people.count > 0 %}
  <h2>Список пользователей</h2>
  <table>
    <thead><th>Id</th><th>Имя</th><th>Возраст</th><th></th></thead>
    {% for person in people %}
    <tr>
      <td>{{ person.id }}</td>
      <td>{{ person.name }}</td>
      <td>{{ person.age }}</td>
      <td><a href="edit/{{ person.id }}">Изменить</a> | <a
href="delete/{{ person.id }}">Удалить</a></td>
    </tr>
    {% endfor %}
  </table>
  {% endif %}
</body>
</html>

```

Чтобы не вводить вручную адреса для редактирования и удаления объектов изменим шаблон **index.html**, где выводится список объектов, добавив в него необходимые ссылки.



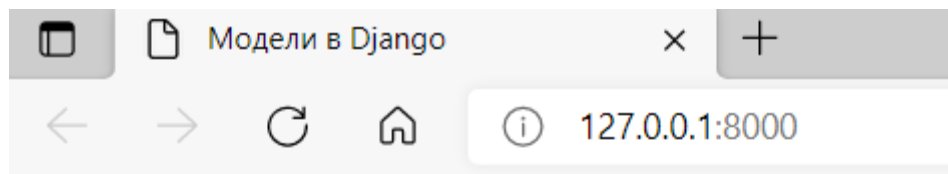
В файле **urls.py** сопоставим функциям **edit()** и **delete()** маршруты:

```
"""
from django.contrib import admin
from django.urls import path, re_path
from artist import views
from django.views.generic import TemplateView

urlpatterns = [
    path('', views.index),
    path('create/', views.create),
    path('edit/<int:id>/', views.edit),
    path('delete/<int:id>/', views.delete),
]
```



перейдем на сервер:



Введите имя

Введите возраст

Сохранить

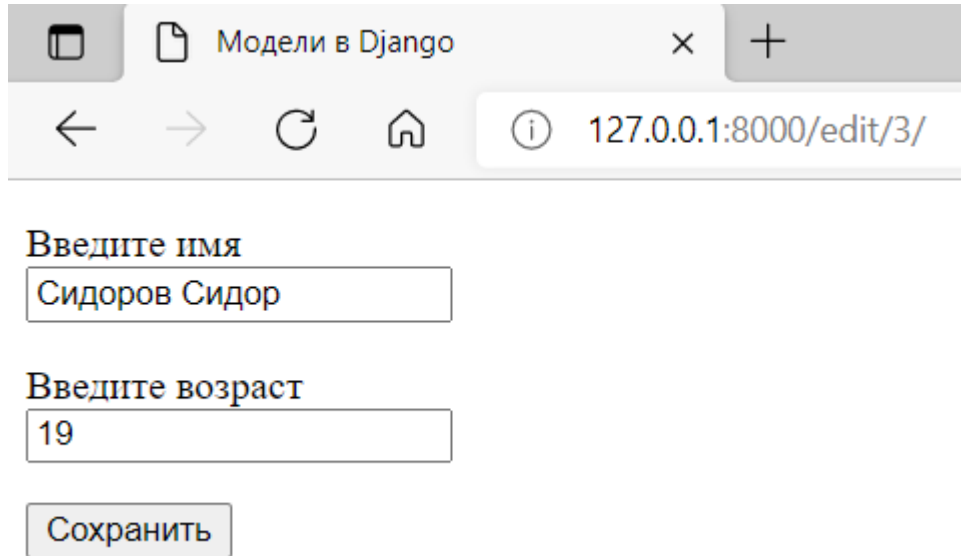
Список пользователей

Id	Имя	Возраст	
1	Иванов Иван	18	Изменить Удалить
2	Петров Петр	23	Изменить Удалить
3	Сидоров Сидор	19	Изменить Удалить
4	Орлов Владимир	35	Изменить Удалить
5	Соколов Игорь	17	Изменить Удалить

Изменим данные в записи с Id=3



Нажмем на кнопку **Изменить**, например, в строке 3 (id=3), в форме отредактируем значения выбранной записи и нажмем кнопку **Сохранить**:



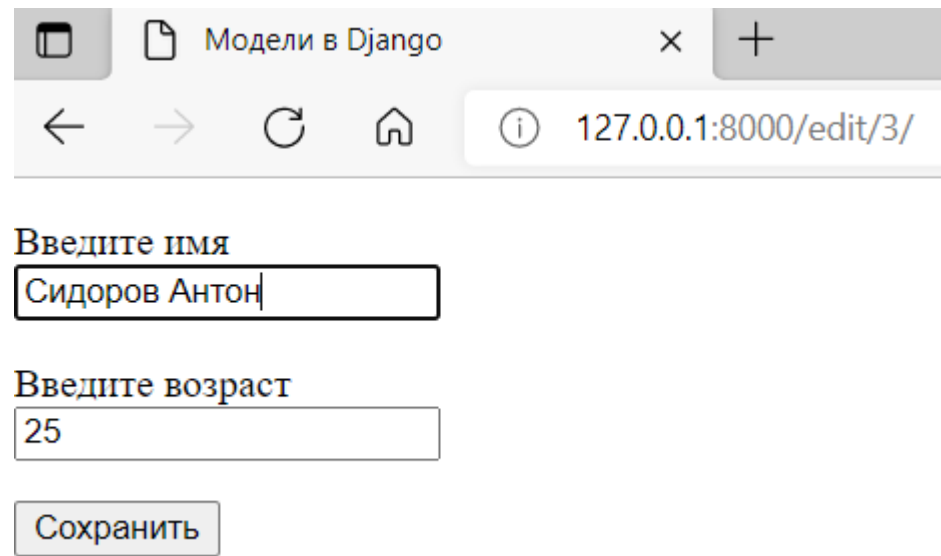
Модели в Django × +

← → ↻ 🏠 ⓘ 127.0.0.1:8000/edit/3/

Введите имя

Введите возраст

Сохранить



Модели в Django × +

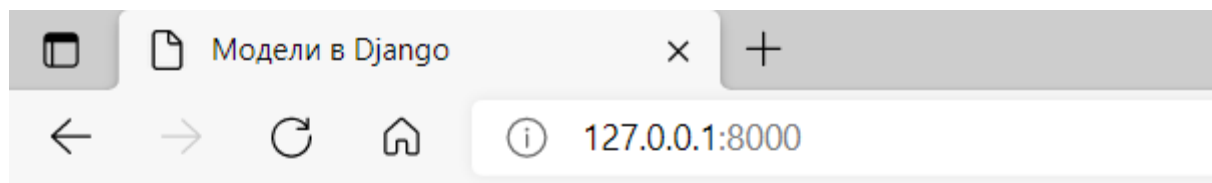
← → ↻ 🏠 ⓘ 127.0.0.1:8000/edit/3/

Введите имя

Введите возраст

Сохранить





Введите имя

Введите возраст

Сохранить

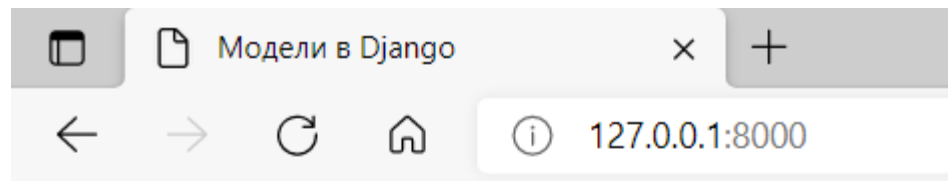
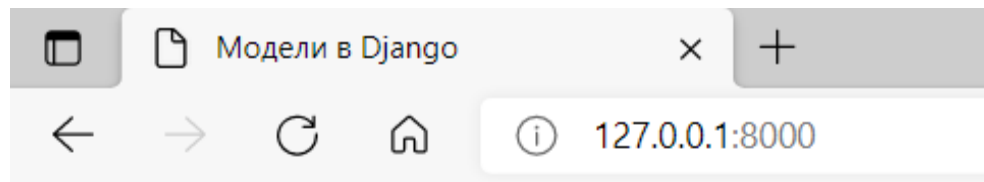
Список пользователей

Id	Имя	Возраст	
1	Иванов Иван	18	Изменить Удалить
2	Петров Петр	23	Изменить Удалить
3	Сидоров Антон	25	Изменить Удалить
4	Орлов Владимир	35	Изменить Удалить
5	Соколов Игорь	17	Изменить Удалить

Измененные данные в записи с Id=3



по кнопке Удалить справа от записи удалим запись, например, с номером 5:



Введите имя

Введите возраст

Сохранить

Список пользователей

Id	Имя	Возраст	
1	Иванов Иван	18	Изменить Удалить
2	Петров Петр	23	Изменить Удалить
3	Сидоров Антон	25	Изменить Удалить
4	Орлов Владимир	35	Изменить Удалить
5	Соколов Игорь	17	Изменить Удалить

Введите имя

Введите возраст

Сохранить

Список пользователей

Id	Имя	Возраст	
1	Иванов Иван	18	Изменить Удалить
2	Петров Петр	23	Изменить Удалить
3	Сидоров Антон	25	Изменить Удалить
4	Орлов Владимир	35	Изменить Удалить



Связи между таблицами базы данных. Виды связей.

Реляционная база данных, спроектированная для реальной задачи, редко состоит из одной таблицы. Как правило, данные конкретной предметной области в процессе проектирования и нормализации распределяются по нескольким таблицам.

Таблицы связываются между собой по ключевым атрибутам, и при этом возникают разные виды связей (отношений):

- «один ко многим»,
- «многие ко многим»,
- «один к одному».



Что означают эти связи?

Связь «один ко многим» - каждой записи в одной таблице (родительской) соответствует несколько записей в другой таблице (дочерней).

Например, есть две таблицы: Отдел и Сотрудник.

Одной записи в таблице Отдел соответствуют несколько записей в таблице Сотрудник.

Что значит “соответствуют”? В каждой записи таблицы Сотрудник, кроме атрибута “Номер сотрудника” - уникальный идентификатор, первичный ключ, **primary key (PK)**, имеется атрибут “Номер отдела” – внешний ключ, мигрирующий ключ, **foreign key (FK)**.

По атрибуту “Номер отдела” таблица Отдел связывается с таблицей Сотрудник.

Аналогично устанавливаются связи **«многие ко многим»** и **«один к одному»**.



Другой пример.

Есть модель (таблица) **Company**, которая представляет производителей и является главной (родительской) таблицей, и есть модель (таблица) **Product**, которая представляет товары разных производителей и является зависимой (дочерней) таблицей.

При создании связи между этими таблицами атрибут первичного ключа **id** главной таблицы **Company** мигрирует в зависимую таблицу **Product** и становится в ней атрибутом внешнего ключа **company.id**. Между таблицами устанавливается **связь один-ко-многим**: одной записи главной таблицы **Company** - производителю в зависимой таблице **Product** соответствует несколько записей -товаров.

Значность этой связи следующая: на стороне атрибута первичного ключа таблицы **Company** стоит 1, на стороне атрибута внешнего ключа в таблице **Product** – много (знак бесконечность).



Дополним файл models.py классами **Company** и **Product**:

```
models x
1  from django.db import models
2
3  # Create your models here.
4
5
6  class Person(models.Model):
7      name = models.CharField(max_length=20)
8      age = models.IntegerField()
9
10
11  class Company(models.Model):
12      name = models.CharField(max_length=30)
13
14
15  class Product(models.Model):
16      company = models.ForeignKey(Company, on_delete=models.CASCADE)
17      name = models.CharField(max_length=30)
18      price = models.IntegerField()
19
```



Рассмотрим установление связи один-ко-многим между таблицами **Company** и **Product**:

```
models x
1  from django.db import models
2
3  # Create your models here.
4
5
6  class Person(models.Model):
7      name = models.CharField(max_length=20)
8      age = models.IntegerField()
9
10
11 class Company(models.Model):
12     name = models.CharField(max_length=30)
13
14
15 class Product(models.Model):
16     company = models.ForeignKey(Company, on_delete=models.CASCADE)
17     name = models.CharField(max_length=30)
18     price = models.IntegerField()
19
```

В приложении Django связь **один-ко-многим** настраивается с помощью конструктора **models.ForeignKey**.

Первый параметр конструктора **models.ForeignKey** указывает, с какой моделью будет создаваться связь - в данном случае это **модель Company**.

Второй параметр - **on_delete** задает опцию удаления объекта текущей модели при удалении связанного объекта главной модели. Это правило отвечает за целостность базы по ссылкам (**Referential integrity**).



для параметра **on_delete** можно использовать следующие значения:

models.CASCADE: автоматически удаляет строку из зависимой таблицы, если удаляется связанная строка из главной таблицы

models.PROTECT: блокирует удаление строки из главной таблицы, если с ней связаны какие-либо строки из зависимой таблицы

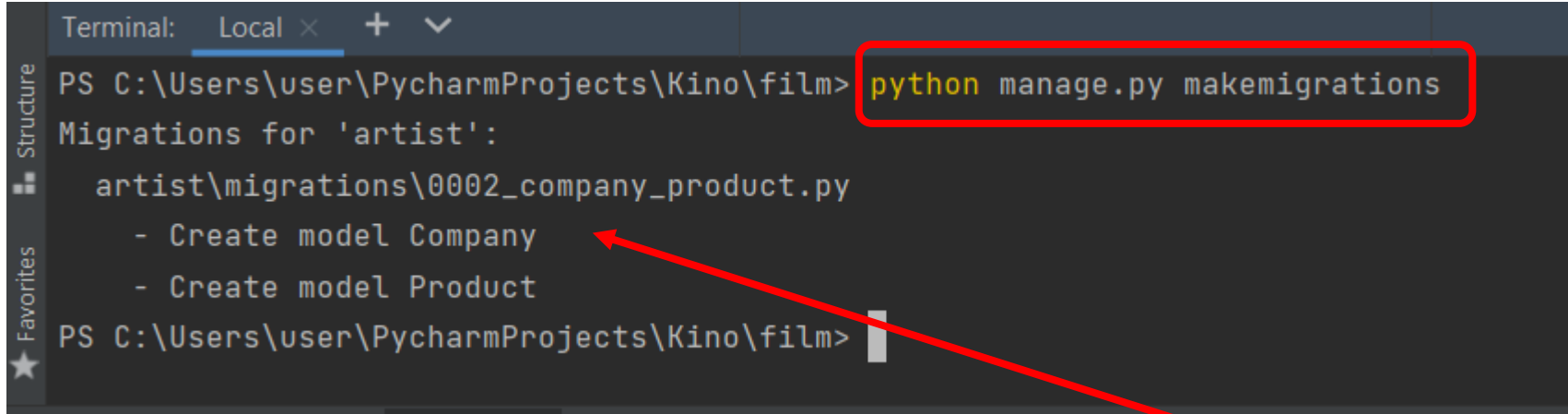
models.SET_NULL: устанавливает NULL при удалении связанной строки из главной таблицы

models.SET_DEFAULT: устанавливает значение по умолчанию для внешнего ключа в зависимой таблице. В этом случае для данного столбца должно быть задано значение по умолчанию

models.DO_NOTHING: при удалении связанной строки из главной таблицы не производится никаких действий в зависимой таблице



Выполним миграцию: `python manage.py makemigrations`

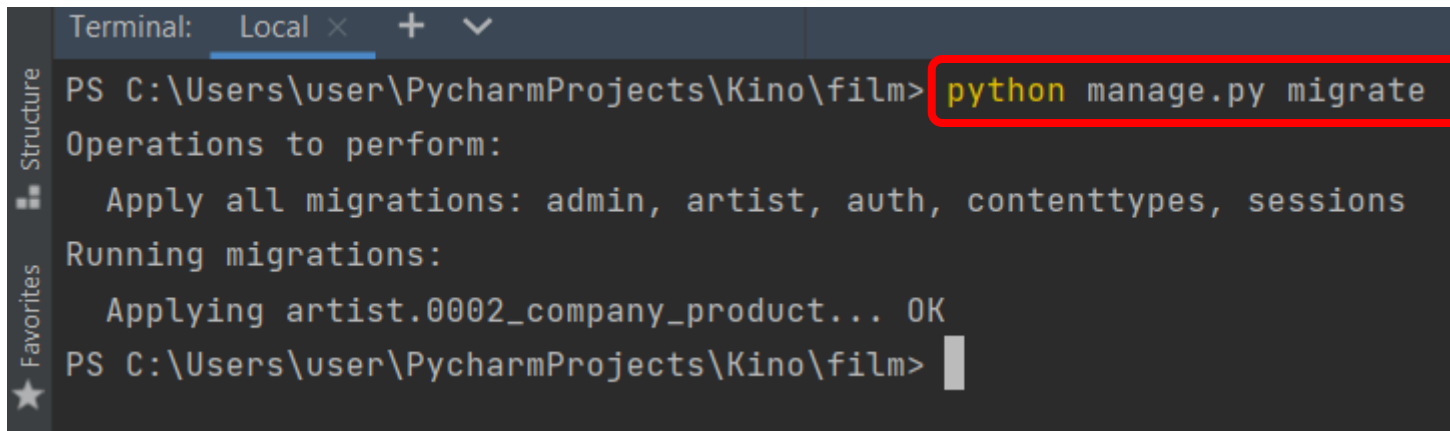


```
Terminal: Local x + v
PS C:\Users\user\PycharmProjects\Kino\film> python manage.py makemigrations
Migrations for 'artist':
  artist\migrations\0002_company_product.py
    - Create model Company
    - Create model Product
PS C:\Users\user\PycharmProjects\Kino\film>
```

A red box highlights the command `python manage.py makemigrations`. A red arrow points from the box to the output line `artist\migrations\0002_company_product.py`.

После выполнения команды будет создан файл миграции `artist\migrations\0002_initial.py`

Выполним второй шаг: `python manage.py migrate`



```
Terminal: Local x + v
PS C:\Users\user\PycharmProjects\Kino\film> python manage.py migrate
Operations to perform:
  Apply all migrations: admin, artist, auth, contenttypes, sessions
Running migrations:
  Applying artist.0002_company_product... OK
PS C:\Users\user\PycharmProjects\Kino\film>
```

A red box highlights the command `python manage.py migrate`.



В результате миграции в базе данных SQLite будут созданы следующие таблицы:

```
CREATE TABLE `artist_company` (  
    `id` integer NOT NULL PRIMARY KEY AUTOINCREMENT,  
    `name` varchar(30) NOT NULL  
);  
  
CREATE TABLE `artist_product` (  
    `id` integer NOT NULL PRIMARY KEY AUTOINCREMENT,  
    `name` varchar(30) NOT NULL,  
    `price` integer NOT NULL,  
    `company_id` integer NOT NULL,  
    FOREIGN KEY(`company_id`) REFERENCES `artist_company`(`id`) DEFERRABLE INITIALLY DEFERRED  
);
```



artist_product (db1)

Структура Данные Ограничения Индексы Триггеры DDL

db1 Имя таблицы: artist_product ☐ WITHOUT ROWID

	Имя	Тип данных	Первичный ключ	Внешний ключ	Уникальность	Проверка	Не NULL	Сравнение	Generate
1	id	integer							
2	name	varchar (30)							
3	price	integer							
4	company_id	bigint							

Базы данных

Фильтр по имени

- > db (SQLite 3)
- > db0 (SQLite 3)
- ▼ db1 (SQLite 3)
 - ▼ Таблицы (13)
 - > artist_company
 - > artist_person
 - > **artist_product**
 - > auth_group
 - > auth_group_permissions
 - > auth_permission
 - > auth_user
 - > auth_user_groups
 - > auth_user_user_permissions
 - > django_admin_log
 - > django_content_type
 - > django_migrations
 - > django_session
 - Представления



The screenshot shows the SQLiteStudio 3.3.3 interface. The main window displays two database windows side-by-side.

The top window, titled "artist_product (db1)", shows the "Данные" (Data) tab. It displays a table with the following data:

	id	name	price	company_id
1	6	iPhone 6.0	65	1
2	5	iPhone 7.1	50	3
3	4	tovar 1	145	2
4	3	tobar 25	75	2
5	2	tovar 34	80	1
6	1	iPhone zz	120	1

The bottom window, titled "artist_company (db1)", shows the "Данные" (Data) tab. It displays a table with the following data:

	id	name
1	3	Apple
2	2	Huawei
3	1	Simens



Операции с моделями

Таблица Product связана с таблицей Company через ключевой атрибут **company_id**.

Но в определении модели Product нет поля **company_id**, а есть только поле **company**, и именно через него в приложении можно получить связанные данные:

получить id компании, которая производит продукт

```
id_company = Product.objects.get(id=1).company.id
```

получить название компании, которая производит продукт

```
name_company = Product.objects.get(id=1).company.name
```

получить перечень товаров, которые производятся компанией "Заря"

```
Product.objects.filter(company__name="Заря")
```

С помощью выражения
модель__свойство

(**внимание! два символа подчеркивания!!!**)

можно использовать свойство главной модели для фильтрации объектов (записей) зависимой модели.



```
from .models import Company, Product
```

```
firma = Company.objects.get(name="Apple")
```

```
# получение всех товаров фирмы "Apple"
```

```
товар = firma.product_set.all()
```

```
# получение количества товаров фирмы "Apple"
```

```
kol_tovar = firma.product_set.count()
```

```
# получение товаров, название которых начинается на "iPhone"
```

```
товар = firma.product_set.filter(name__startswith="iPhone")
```

С точки зрения модели `Company` она не имеет никаких свойств, которые связывали бы ее с моделью `Product`, но с помощью команды, имеющей синтаксис:

"главная_модель"."зависимая_модель"_set
можно по записи из главной модели получать связанные записи в зависимой модели.



С помощью выражения **_set** можно выполнять операции добавления, изменения, удаления объектов зависимой модели из главной модели. Например:

создаем объект Company с именем Apple

```
firma = Company.objects.create(name="Apple")
```

создание товара компании Apple

```
firma.product_set.create(name="iPhone 8", price=67890)
```

отдельное создание объекта с последующим добавлением в БД:

```
ipad = Product(name="iPad", price = 34560)
```

при добавлении необходимо указать параметр bulk =False

```
firma.product_set.add(ipad, bulk = False)
```



исключает из компании все товары,

при этом товары остаются в базе данных и не привязаны к компании

работает, если в зависимой модели ForeignKey(Company, null = True)

```
firma.product_set.clear()
```

то же самое, только в отношении одного объекта

```
ipad = Product.objects.get(name="iPad")
```

```
firma.product_set.remove(ipad)
```



Отметим три метода: `add()`, `clear()`, `remove()`, которые могут быть использованы в сочетании с `_set`:

Метод `add()`: добавляет как саму запись в дочернюю таблицу, так и связь между объектом зависимой модели и объектом главной модели.

В своей сути метод **`add()`** фактически вызывает для модели еще и метод **`update()`** для добавления связи. Однако это требует, чтобы обе модели уже были в базе данных.

И здесь применяется параметр **`bulk = False`**, чтобы объект зависимой модели сразу был добавлен в БД и для него была установлена связь.



Метод clear(): удаляет связь между всеми объектами зависимой модели и объектом главной модели.

При этом сами объекты зависимой модели остаются в базе данных, и для их внешнего ключа устанавливается значение NULL.

Поэтому данный метод будет работать, если в самой зависимой модели при установке связи использовался параметр **null = True**: **ForeignKey(Company, null = True)**.

Метод remove(): также, как и **clear()** удаляет связь, только между одним объектом зависимой модели и объектом главной модели.

При этом также все объекты дочерней таблицы остаются в базе данных.

И также в самой зависимой модели при установке связи должен использоваться параметр **null = True**.



Связь вида многие-ко-многим (Many to Many). Создание связи.

Примеры связи многие-ко-многим:

Пациент – Доктор.

Каждый пациент может посетить много докторов, и каждый доктор может принять много пациентов.

Пациент – Диагноз.

Каждый пациент может иметь много диагнозов, и каждый диагноз может быть поставлен многим пациентам.

Курс – Студент

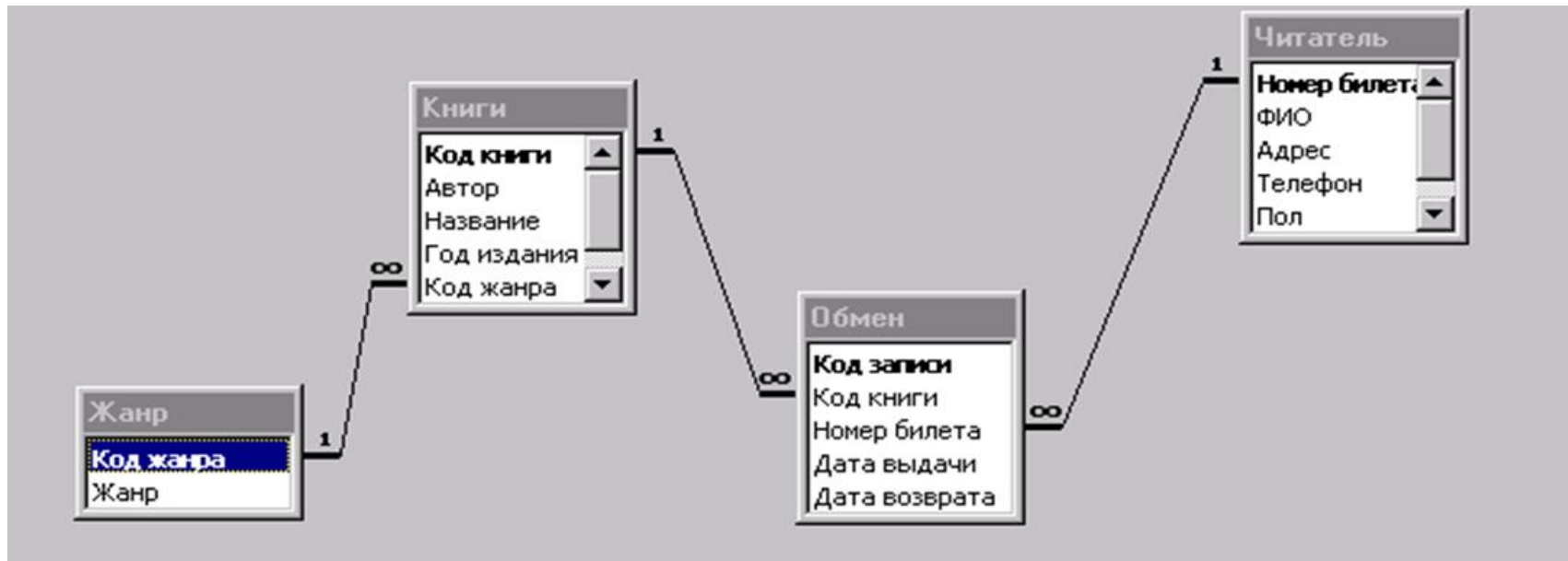
Каждый студент может посетить много курсов, и каждый курс читается многим студентам.



Связь **многие-ко-многим** допустима только на уровне логической модели.

При реализации модели в конкретной реляционной СУБД связь **многие-ко-многим** преобразуется в связь **один-ко-многим** путем создания третьей таблицы, в которую мигрируют ключевые атрибуты первых двух.

Например, в случае **Книга – Читатель** создается третья таблица **Обмен**, в которую мигрируют внешние ключи – **Код книги** из таблицы **Книга** и **Номер билета** из таблицы **Читатель**:



Для создания отношения **многие-ко-многим** между таблицами **Course** и **Student** применяется конструктор **ManyToManyField**.

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

```
class Course(models.Model):
```

```
    name = models.CharField(max_length=30)
```

```
class Student(models.Model):
```

```
    name = models.CharField(max_length=30)
```

```
    courses = models.ManyToManyField(Course)
```



```
CREATE TABLE `artist_course` (  
    `id` integer NOT NULL PRIMARY KEY AUTOINCREMENT,  
    `name` varchar(30) NOT NULL  
);  
  
CREATE TABLE `artist_student` (  
    `id` integer NOT NULL PRIMARY KEY AUTOINCREMENT,  
    `name` varchar(30) NOT NULL  
);
```

В конструктор `models.ManyToManyField` передается сущность, с которой устанавливается отношение **многие-ко-многим**.

В результате будет создаваться промежуточная таблица, через которую будут связаны две исходных.

В результате миграции в базе данных SQLite будут созданы три таблицы.

Третья таблица "`artist_student_courses`" выступает в качестве связующей таблицы.



```
CREATE TABLE `artist_student_courses` (  
    `id` integer NOT NULL PRIMARY KEY AUTOINCREMENT,  
    `student_id` integer NOT NULL,  
    `course_id` integer NOT NULL,  
  
    FOREIGN KEY(`student_id`) REFERENCES `artist_student`(`id`) DEFERRABLE INITIALLY  
    DEFERRED,  
  
    FOREIGN KEY(`course_id`) REFERENCES `artist_course`(`id`) DEFERRABLE INITIALLY DEFERRED  
• );
```



Операции с моделями

Через свойство `courses` в модели `Student` мы можем получать связанные со студентом курсы и управлять ими.

`# создадим студента`

```
ivan = Student.objects.create(name="Ivan")
```

`# создадим один курс и добавим его в список курсов Ивана`

```
ivan.courses.create(name="Algebra")
```



получим все курсы студента

```
courses = Student.objects.get(name=" Ivan ").courses.all()
```

получаем всех студентов, которые посещают курс Алгебра

```
students = Student.objects.filter(courses__name="Algebra")
```

В случае, где производится фильтрация студентов по посещаемому курсу, для передачи в метод **filter** названия курса используется параметр, название которого начинается с названия свойства, через которое идет связь со второй моделью.

И далее через два знака подчеркивания указывается имя свойства второй модели, например, **courses__name** или **courses__id**.

То есть, можно получить данные о курсах студента через свойство **courses**, которое определено в модели **Student**.



создадим курс:

```
kurs_python = Course.objects.create(name="Python")
```

создаем студента и добавляем его на курс:

```
kurs_python.student_set.create(name="Bob")
```

получим всех студентов курса:

```
students = kurs_python.student_set.all()
```

удаляем с курса одного студента:

```
s_bob = Student(name="Bob")
```

```
kurs_python.student_set.remove(s_bob)
```



удаляем с курса всех студентов:

```
kurs_python.student_set.clear()
```

отдельно создаем студента и добавляем его на курс

```
sam = Student(name="Sam")
```

```
sam.save()
```

```
kurs_python.student_set.add(sam)
```

получим количество студентов по курсу

```
number = kurs_python.student_set.count()
```



Связь вида один-к-одному (One to one). Создание связи.

Отношение **один-к-одному** предполагает, что каждая запись из одной таблицы может быть связана только с одной записью в другой таблице.

Например, пользователь может иметь какие-либо данные, которые описывают его учетные данные.

Всю базовую информацию о пользователе, типа имени, возраста, можно выделить в одну модель, а учетные данные - логин, пароль, время последнего входа в систему, количество неудачных входов и т.д. - в другую модель:



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

```
class User(models.Model):
```

```
    name = models.CharField(max_length=20)
```

```
class Account(models.Model):
```

```
    login = models.CharField(max_length=20)
```

```
    password = models.CharField(max_length=20)
```

```
    user = models.OneToOneField(User, on_delete = models.CASCADE, primary_key = True)
```



Для создания отношения один к одному применяется конструктор `models.OneToOneField()`.

Первый параметр конструктора указывает, с какой моделью будет ассоциирована данная сущность (в данном случае ассоциация с моделью `User`).

Второй параметр `on_delete = models.CASCADE` говорит, что данные текущей модели (`Account`) будут удаляться в случае удаления связанного объекта главной модели (`User`) – правило ссылочной целостности данных.

Третий параметр `primary_key = True` указывает, что внешний ключ (через который идет связь с главной моделью) в то же время будет выступать и в качестве первичного ключа.



В результате миграции в базе данных SQLite будут созданы следующие таблицы:

```
CREATE TABLE `artist_user` (  
    `id` integer NOT NULL PRIMARY KEY AUTOINCREMENT,  
    `name` varchar(20) NOT NULL  
);
```

```
CREATE TABLE `artist_account` (  
    `login` varchar(20) NOT NULL,  
    `password` varchar(20) NOT NULL,  
    `user_id` integer NOT NULL,  
    PRIMARY KEY(`user_id`),  
    FOREIGN KEY(`user_id`) REFERENCES `artist_user`(`id`) DEFERRABLE INITIALLY DEFERRED  
);
```



Операции с моделями

создадим пользователя

```
sam = User.objects.create(name="Sam")
```

создадим аккаунт пользователя Sam

```
acc = Account.objects.create(login = "1234", password = "5678", user = sam)
```

изменяем имя пользователя

```
acc.user.name = "Bob"
```

сохраняем изменения в базе данных

```
acc.user.save()
```



При этом через модель **User** мы также можем оказывать влияние на связанный объект **Account**.

Несмотря на то, что явным образом в модели **User** определено только одно свойство - **name**, при связи типа **один-к-одному** неявно создается еще одно свойство, которое называется по имени зависимой модели и которое указывает на связанный объект этой модели.

В данном случае это свойство будет называться "**account**":



создадим пользователя:

```
tom = User.objects.create(name="Tom")
```

создадим аккаунт пользователя:

```
acc = Account(login = "1234", password="6789")
```

```
tom.account = acc
```

```
tom.account.save()
```

обновляем данные

```
tom.account.login = "alfa"
```

```
tom.account.password = "123456"
```

```
tom.account.save()
```



Конец части 3. Продолжение следует...

