Django ORM: модели, миграции, CRUD, выгрузка в JSON | Я.Шпора

Модели Django обычно хранят в файлах *models.py* в папках приложений. Модели наследуются от класса `Model` из модуля `models`.

```
# Импорт модуля с классом Model, от этого класса наследуются модели:
from django.db import models

class VideoProduct(models.Model):
    title = models.CharField(max_length=128)
```

Типы полей

Тип поля модели указывается при помощи специальных классов. Вот несколько популярных типов (в скобках — название аналогичного типа в SQL):

`models.IntegerField()` — натуральное число (INTEGER);

`models.FloatField()` — число с плавающей точкой (REAL);

`models.BooleanField()` — булев тип `False` / `True` (BOOL);

`models.CharField()` — строка, текстовое поле с ограничением по числу символов (VARCHAR);

`models.TextField()` — текстовое поле (TEXT);

`models.DateField()` — дата, как `datetime.date` в Python (DATE);

`models.DateTimeField()` — дата и время, как `datetime.datetime` в Python (DATATIME);

`models.SlugField()` — «слаг», строка, содержащая только цифры, буквы латиницы и символы `-` и `_`. Обычно слаг используют для создания человекочитаемых URL;

`models.ImageField()` — изображения.

Связь 1:1

При связи «один-к-одному» каждая запись одной таблицы БД может быть связана только с одной из записей другой таблицы.

Для описания связи «один к одному» в моделях используется тип поля `models.OneToOneField()`.

```
# models.py
class OriginalTitle(models.Model):
    title = models.CharField(max_length=128)

class VideoProduct(models.Model):
    title = models.CharField(max_length=128)
    # Описываем поле, ссылающееся на модель OriginalTitle:
    original_title = models.OneToOneField(
        # На какую модель ссылаемся:
        OriginalTitle,
        # Поведение при удалении:
        # если связанный объект модели OriginalTitle будет удалён,
        # то и объект класса VideoProduct будет удалён.
        on_delete=models.CASCADE
    )
```

`on_delete` — обязательный параметр для ссылающихся полей (`OneToOneField`, `ForeighnKey`, `ManyToManyField`).

Параметр `on_delete` может принимать разные значения:

- `on_delete=models.CASCADE` — если удаляется запись `OriginalTitle`, то будет удалена и ссылающаяся на неё запись из таблицы `VideoProduct`.
- `on_delete=models.SET_NULL` — при удалении объекта, на который ведёт ссылка, в ссылающихся записях вместо ссылки на объект будет установлен `NULL`.
- `on_delete=models.SET_DEFAULT` — при удалении объекта, на который ведёт ссылка, в ссылающихся записях вместо ссылки на объект будет установлено значение по умолчанию, указанное в параметре `default` поля ссылки.

Для `on_delete` есть и другие аргументы: `PROTECT`, `RESTRICT` и `DO_NOTHING`.

Связь N:1

Связь «многие-к-одному» позволяет связать несколько записей одной таблицы с одной и той же записью другой таблицы.

В моделях связь «многие-к-одному» указывается в поле типа `ForeignKey`:

```
# models.py
from django.db import models

class ProductType(models.Model):
    title = models.CharField(max_length=128)

class VideoProduct(models.Model):
    ...
    product_type = models.ForeignKey(
        ProductType,
        on_delete=models.CASCADE
    )
```

В описании связи обязательно должен быть указан аргумент `on_delete`.

Связь N:M

Связь «многие-ко-многим» позволяет связать каждую запись первой таблицы с несколькими записями второй, а каждую запись второй таблицы — с несколькими записями первой.

В Django есть разные варианты для создания связи «многие-ко-многим».

Через промежуточную модель, созданную вручную:

```
# models.py
from django.db import models

class VideoProduct(models.Model):
    title = models.CharField(max_length=128)

class Director(models.Model):
    full_name = models.CharField(max_length=128)

# Промежуточная модель: в ней указываются связи.
class DirectorVideoProduct(models.Model):
```

```
    video_product = models.ForeignKey(VideoProduct, on_delete=models.CA
SCADE)
    director = models.ForeignKey(Director, on_delete=models.CASCADE)
```

Через поле типа `models.ManyToManyField()`:

```
# models.py
from django.db import models

class Director(models.Model):
    full_name = models.CharField(max_length=128)

class VideoProduct(models.Model):
    title = models.CharField(max_length=128)
    directors = models.ManyToManyField(Director)  # Поле N:M.
```

При использовании поля `ManyToManyField()` автоматически создаётся промежуточная таблица, которая реализует связь «многие ко многим».

При использовании поля `ManyToManyField()` промежуточную таблицу тоже можно создать вручную:

- описывается промежуточная модель с нужными полями;
- промежуточная модель указывается в параметре `through` поля `ManyToManyField()`:

```
# models.py
from django.db import models

class Director(models.Model):
    full_name = models.CharField(max_length=128)

class VideoProduct(models.Model):
    title = models.CharField(max_length=128)
    # Параметр through указывает, какую модель надо назначить промежуто
чной:
    directors = models.ManyToManyField(Director, through='Partnership')

# Промежуточная модель:
```

```
class Partnership(models.Model):
    # Поле, ссылающееся на модель Director:
    director = models.ForeignKey(Director, on_delete=models.CASCADE)
    # Поле, ссылающееся на модель VideoProduct:
    videoproduct = models.ForeignKey(VideoProduct, on_delete=models.CAS
CADE)
    # Дополнительные поля:
    # дата начала работы режиссёра над фильмом...
    date_joined = models.DateField()
    # ...и история о том, почему на фильм пригласили именно этого режис
сёра.
    invite_reason = models.CharField(max_length=300)
```

Имя для обратной связи между таблицами

Для объектов, на которые ссылаются поля типа `ForeignKey`, ORM создаёт специальный интерфейс, через который можно получить доступ к объектам исходной модели: если модель `First` ссылается на модель `Second`, то из модели `Second` тоже можно обратиться к модели `First`.

Для создания такого «интерфейса обратной связи» используется опциональный параметр `related_name`.

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(
                Author,
                on_delete=models.CASCADE,
                related_name='books'
        )
```

Если какой-то объект модели `Book` ссылается на объект модели `Author`, то при установленном параметре `related_name` из объекта `Author` можно получить ссылающиеся на него объекты `Book`:

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    # Поле author ссылается на объект модели Author (на автора книги):
    author = models.ForeignKey(Author, on_delete=models.CASCADE, relate
d_name='books')
```

Пример применения `related_name`:

```
# Получаем объект автора с заданным значением поля name:
author = Author.objects.get(name='Александр Пушкин')
# По related_name books получаем все объекты, ссылающиеся на полученный
объект автора:
pushkin_books = author.books.all()
```

Если разработчик не указал явным образом в модели значение параметра `related_name`, это значение будет сгенерировано автоматически — из названия модели и суффикса `_set`. Таким образом, для приведённого примера `related_name` было бы `book_set`.

```
author = Author.objects.get(name='Александр Пушкин')
pushkin_books = author.book_set.all()
```

Абстрактные модели

Абстрактные модели не создают таблиц в БД. Чтобы объявить модель абстрактной, необходимо во вложенном классе `Meta` объявить атрибут `abstarct` со значение `True`.

```
class BaseModel(models.Model):
    """
    Абстрактная модель.
    Добавляет к модели дату создания и последнего изменения.
    """
```

```
    created_at = models.DateTimeField(auto_now_add=True)
    modified_at = models.DateTimeField(auto_now_add=False, auto_now=Tru
e)
    # С помощью необязательного внутреннего класса Meta можно добавить
    # к модели дополнительные настройки.
    class Meta:
        # Эта строка объявляет модель абстрактной:
        abstract = True
```

Подключение базы данных к Django

Настройки подключения к БД в Django-проекте указываются в константе `DATABASES` в *settings.py*:

```
# <название проекта>/settings.py

DATABASES = {
    'default': {
        # К проекту по умолчанию подключена СУБД SQLite:
        'ENGINE': 'django.db.backends.sqlite3',
        # Файл с базой данных находится в одной папке с manage.py.
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

Миграции

Миграции — это процесс автоматического создания и применения изменений в базе данных на основе моделей приложения.

> **Файлы миграции каждого приложения следует добавлять в git: миграции — это неотъемлемая часть проекта.**

Команды:

- `python manage.py makemigrations` — создание новых миграций на основе изменений, внесённых в модели.
- `python manage.py migrate` — применение миграций.

- `python manage.py sqlmigrate <имя приложения> <номер миграции>` — отображение SQL-запросов, которые будут отправлены при миграции.

CRUD-операции

Операции с реляционными базами данных делят на четыре группы:

- **Create** — создание записей.
- **Read** — чтение записей.
- **Update** — изменение записей.
- **Delete** — удаление записей.

Все примеры будут основаны на модели `Category`:

```
from django.db import models

class Category(models.Model):
    title = models.CharField(max_length=256)
    slug = models.SlugField(max_length=64, unique=True)
    output_order = models.PositiveSmallIntegerField(default=100)
```

Create

Создать новую запись в БД и вернуть объект модели: метод `.create()`.

```
>>> Category.objects.create(
    title='Категория, созданная через shell',
    slug='shell_category'
)
<Category: Category object (1)>
```

Read

Получить все объекты модели: метод `.all()`.

```
>>> Category.objects.all()
<QuerySet [<Category: Category object (1)>, <Category: Category object
(2)>]>
```

Поиск объектов по заданным признакам: метод `.filter()`.

```
# Получить объект, у которого поле slug содержит значение 'shell_catego
гу':
>>> Category.objects.filter(slug='shell_category')
<QuerySet [<Category: Category object (1)>]>
```

Получить отдельный объект из БД: метод `.get()`.

```
>>> Category.objects.get(pk=1)  # Получить объект, у которого Primary K
ey равен 1
<Category: Category object (1)>
```

Update

Изменить объект можно двумя способами.

1. Получить коллекцию объектов и присвоить одному или нескольким полям этих объектов новые значения:

   ```
   # Category.objects.all() возвращает QuerySet со всеми объектами мод
   ели,
   # а метод update() меняет свойства всех объектов:
   >>> Category.objects.all().update(title='Изменённое поле категори
   и', is_published=True)
   # В ответ получим количество изменённых записей:
   2
   ```

2. Получить один объект, присвоить новое значение одному из его полей и вызвать метод `.save()`:

   ```
   # Получаем объект и сохраняем его в переменную category_for_change:
   >>> category_for_change = Category.objects.get(pk=1)
   # Меняем значение одного из полей:
   >>> category_for_change.title = 'Ещё раз изменённое поле категории'
   >>> category_for_change.is_published = False
   # Новое значение присвоено объекту модели, но в БД всё ещё хранится
   старое значение.
   # Чтобы сохранить новое значение в базе данных — вызываем метод sav
   e():
   ```

```
>>> category_for_change.save()

# Смотрим информацию из обновлённых полей:
>>> Category.objects.get(pk=1).title
'Ещё раз изменённое поле категории'
>>> Category.objects.get(pk=1).is_published
False
```

Delete

Для удаления объектов применяют метод `.delete()`.

```
>>> category_for_delete = Category.objects.get(pk=1)
>>> category_for_delete.delete()
# Будет выведено
(1, {'ice_cream.Category': 1})
```

Удалить набор объектов:

```
# Получаем QuerySet и удаляем все содержащиеся в нём объекты:
Category.objects.all().delete()
```

JSON для наполнения БД и для выгрузки информации

Загрузить данные из файла *db.json* (имя может быть любым) в базу данных:

```
python manage.py loaddata db.json
```

Выгрузить данные из БД в файл *db.json*:

```
python manage.py dumpdata -o db.json
```

Выгрузить данные из приложения **ice_cream** в файл *ice_cream.json*:

```
python manage.py dumpdata ice_cream -o ice_cream.json
```

Сохранить данные из отдельной таблицы в файл *ice_cream_icecream.json*:

```
# Сохраняем данные модели icecream приложения ice_cream:
python manage.py dumpdata ice_cream.icecream -o ice_cream_icecream.json
```

Экспортировать все таблицы за исключением перечисленных (`--exclude`) в файл *without_ice_cream_icecream.json*:

```
# Сохраняем все данные из проекта, кроме данных модели icecream приложе
ния ice_cream:
python manage.py dumpdata --exclude ice_cream.icecream -o without_ice_c
ream_icecream.json
```

При создании фикстур может возникнуть проблема с кодировкой; фикстуры будут созданы, но прочесть кириллицу будет невозможно.

Решением проблемы будет добавление ключа `-Xutf8` для `python`. Например:

```
python -Xutf8 manage.py dumpdata --indent 2 -o indented_db.json
```

Если данные были экспортированы на одном компьютере, а залить базу надо на другом — возникнет проблема: при импорте данных будет выброшено исключение `IntegrityError`.

Чтобы избежать ошибки, при создании фикстуры нужно исключить таблицу `contenttypes`:

```
python manage.py dumpdata --exclude contenttypes -o db.json
```

Проблемы при импорте могут возникнуть и с таблицей `auth.permission`, которая хранит информацию о правах пользователей. Эту таблицу тоже можно исключить из фикстур: `--exclude auth.permission`.