



Django Framework

development backend web apps

For beginners



Урок 3. Django ORM: создание моделей, миграции

Проектирование структуры БД для LMS

Перед тем, как начать непосредственно писать код для создания моделей и производить миграцию в БД, нам необходимо понять, какие данные мы будем хранить в базе для организации системы онлайн-обучения, какие связи установим между таблицами и многое другое.

Первое, над чем стоит задуматься, это хранение пользовательских данных.

Наша система будет хранить следующие характеристики пользователя: имя, фамилию, email, дату рождения (для определения возраста ученика), аватарку к профилю, описание профиля, статус аккаунта (активен / неактивен), статус самого участника, допустим, участник может являться не только учеником, но и автором одного из курсов, и также статус суперпользователя, позволяющий такому участнику производить всевозможные административные операции над базой данных, а также получать доступ к интерфейсу административной панели Django.

В итоге у нас получится такая блок-схема, характеризующая данную таблицу:

User
id (PK)
firstname
lastname
email (UK)
is_active
is_staff
is_superuser
birthday
avatar
description

Поля, помеченные PK и UK, характеризуют уникальный идентификатор записи в таблице (Primary Key) и уникальное поле (Unique Key) соответственно. В данном случае уникальным полем будет являться email, по которому в будущем мы будем производить авторизацию.

Следующей сущностью, информацию о которой необходимо хранить для полноценной организации обучения, являются непосредственно курсы.

Курсы могут иметь следующие характеристики: само название курса, его описание, автор, дата старта, продолжительность / дата конца, цена, количество уроков.

В итоге получается такая схема:

Course
id (PK)
title (UK)
author
description (UK)
start_date
duration
price (DEFAULT 0)
count_lessons

Эта таблица будет содержать 1 уникальное поле: название курса (title). Поле price по умолчанию сделаем равным 0, так как наша система будет предполагать публикацию в том числе и бесплатных курсов. Поле author будет ссылкой на таблицу User, содержащую лишь Primary Key, по которому в дальнейшем мы сможем получить всю информацию об авторе курса.

Следующая таблица будет нужна для хранения перечня уроков для каждого курса. Она будет содержать 3 поля: идентификатор курса, являющийся внешней ссылкой на таблицу Курсы, название урока, краткое описание.

Её схема выглядит так:

Lessons
id (PK)
course (FK)
name (UK)
preview

Поле, помеченное FK, означает Foreign Key, т.е. внешняя ссылка.

Давайте, подумаем... Без какого элемента наша система не будет полноценной LMS?

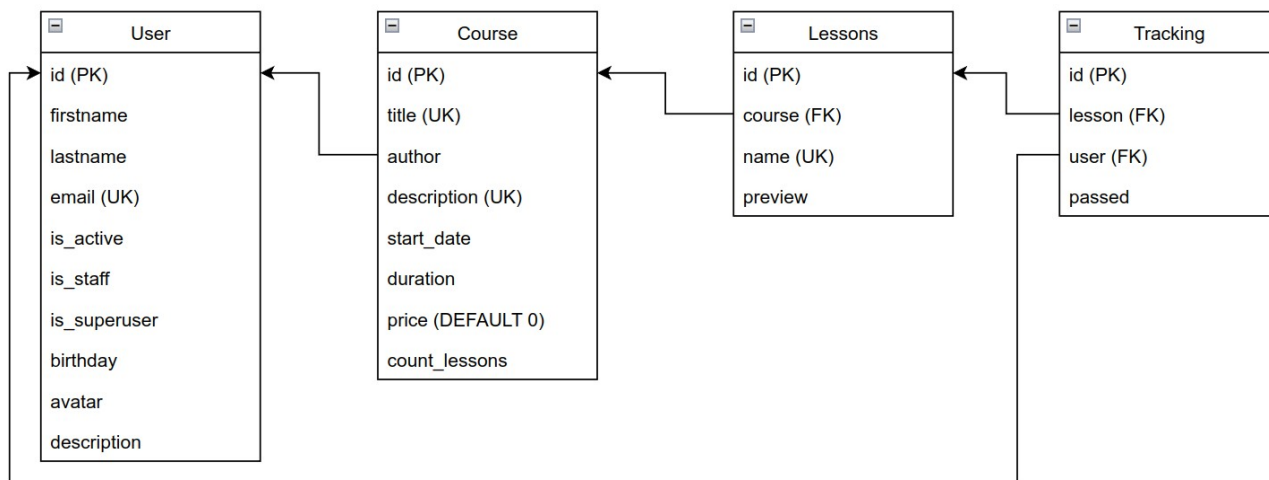
Правильно, нам каким-то образом нужно отслеживать успехи учеников на протяжении прохождения курса. Для этого нам потребуется ещё одна таблица. Назовём ей Трекинг.

Она будет содержать всего 3 поля: идентификатор ученика, идентификатор номера урока курса, статус прохождения. Идентификатор курса нам не нужно создавать, так как номер урока однозначно определяет курс. Мы его сможем получить по ссылке.

Внешний вид схемы отображён ниже:

Tracking
id (PK)
lesson (FK)
user (FK)
passed

Полная же структура хранения данных для нашей LMS будет выглядеть следующим образом:



Стрелками показаны необходимые связи.

Создание моделей

Как было описано в прошлом уроке, для создания моделей необходимо воспользоваться файлом `models.py`.

В начале создадим пользовательскую модель. Поскольку данные будут записываться в БД при регистрации, поэтому создание модели предполагается реализовать в файле `models.py` приложения `auth_app`.

Все создаваемые модели в Django наследуются от базового класса `Model` из модуля `django.db.models`. Имя, которое мы дадим классу, и будет именем базы данных. Поля модели записываются как атрибуты класса и являются объектами типов полей, заранее реализованных Django. Каждое поле имеет свои собственные параметры, с помощью которых и происходит валидация вводимых клиентом данных.

Перечислим основные классы полей:

- ◆ `IntegerField` — 32-разрядное целое число
- ◆ `PositiveIntegerField` — то же целое число, но беззнаковое.

- ◆ **FloatField** — вещественное число (число с плавающей точкой)
- ◆ **DecimalField** — так же вещественное число, но с фиксированным количеством цифр. В нём мы можем указать точное количество цифр во всём числе и количество цифр после запятой (атрибуты `max_digits` и `decimal_places` соответственно). Этот тип поля реализован с использованием стандартного модуля Python `decimal`.
- ◆ **CharField** — символьное поле ограниченной длины. Оно имеет обязательный атрибут `max_length`, позволяющий указать максимальное количество символов в строке. Не рекомендую указывать большое число символов, так как для каждого поля в базе данных будет выделяться полный объем, необходимый для хранения максимального количества символов, даже если в атрибуте Вы указали 32 символа, а ввели лишь 20.
- ◆ **TextField** — это поле, наоборот, не ограничено по длине. Его стоит указывать, когда мы точно не знаем, сколько оно будет занимать. Это поле, указывается, например, для содержимого статьи, ведь оно может быть большой длины.
- ◆ **EmailField** — строковое представление адреса электронной почты. От обычного символьного поля оно отличается лишь тем, что предоставляет реализацию валидации email.
- ◆ **SlugField** — строка-идентификатор записи в таблице, которая передаётся в составе URL-адреса.
- ◆ **BooleanField** — обычное логическое поле, которое может хранить лишь True / False.
- ◆ **DateField** — строковое представление в формате даты, реализованное с помощью Python-модуля `datetime`. Имеет 2 необязательных атрибута типа bool:
 - `auto_now_add` — в поле записывается дата запись только при его создании

- **auto_now** — в поле записывается дата при каждом изменении записи, т. е. и при создании, и при редактировании.

Если указать хотя бы в одном из этих полей True, то поле не будет отображаться в форме для создания записи.

- ◆ **DateTimeField** — аналогично DateField, но в нём также хранится временная метка в виде объекта datetime.
- ◆ **BinaryField** — данные в двоичном виде любой длины. Представляет из себя объект типа bytes.
- ◆ **AutoField** — 32-разрядное автоинкрементное поле.
- ◆ **UUIDField** — уникальный идентификатор в виде строки, являющийся объектом типа UUID модуля uuid.

Все поля также поддерживают следующие атрибуты:

- ◆ **verbose_name** — название поля в понятном для человека виде.
- ◆ **default** — значение поля по умолчанию. Применяется в том случае, если пользователь ничего не ввёл в данное поле.
- ◆ **help_text** — текст-подсказка. Его можно использовать, например, для сообщения пользователю о формате ввода телефона.
- ◆ **unique** — позволяет указать, будет ли данное поле уникальным в пределах таблицы.
- ◆ **null** — указание на то, что строковые поля могут хранить пустые значения. Если указан False, то поле должно содержать хотя бы пустую строку.
- ◆ **blank** — то же самое, что и null, но отличается тем, что позволяет производить валидацию поля сразу в самом браузере при вводе, а не только на уровне базы данных.

- ◆ `db_index` — логический атрибут, позволяющий проиндексировать данное поле в таблице.
- ◆ `primary_key` — позволяет указать ключевое поле таблицы. Такое поле автоматически станет уникальным и обязательным к заполнению. Отметим, что таблица может содержать только одно ключевое поле.
- ◆ `editable` — логический атрибут. С его помощью мы можем указать, выводить ли данное поле в составе HTML-формы.
- ◆ `db_column` — строковое представление поля в таблице на уровне базы данных.

На данный момент ограничимся данными атрибутами и классами полей. В процессе разработки проекта мы узнаем чуть больше.

Применим полученные знания для создания пользовательской модели.

Как говорилось в первом уроке, в Django уже реализована собственная система аутентификации, которая в том числе предоставляет уже готовую пользовательскую модель. Её класс `User` расположен в модуле `django.contrib.auth.models`. Но есть одна проблема, которую необходимо решить. Для нашего проекта класс `User` предоставляет неполный список полей, который нам нужен. Например, в нём нет поля `avatar`, `birthday`, `description`. Поэтому нам нужно расширить возможности данного класса. Этого можно добиться несколькими способами:

- расширить модель пользователя, создав собственную модель, одно поле которой будет представлять связь один-к-одному с моделью `User`
- расширить модель пользователя, наследуя класс `AbstractBaseUser`
- расширить модель пользователя, наследуя класс `AbstractUser`

Каждый из перечисленных способов имеет свои преимущества и недостатки.

В первом случае Django придётся делать дополнительные запросы к связанным данным, что в итоге увеличит нагрузку на БД.

Второй способ считается одним из самых сложных, потому что в нём нам придётся с нуля создавать менеджер пользователей для управления операцией создания пользователя.

Последний же вариант считается самым оптимальным, поскольку класс **AbstractUser** обеспечивает полную реализацию стандартной модели. Он имеет все стандартные поля, нам остаётся только добавить кастомные поля. Также этот способ позволяет нам указать идентификатор для входа пользователя в систему в виде email (в стандартной реализации используется username).

Его мы и будем использовать.

Класс **AbstractUser** нам нужно импортировать всё из того же модуля **django.contrib.auth.models**.

```
1  from django.contrib.auth.models import AbstractUser
2  from django.db import models
3  from .functions import get_timestamp_path_user
4
5
6  class User(AbstractUser):
7      birthday = models.DateField(verbose_name='Дата рождения', blank=False)
8      description = models.TextField(verbose_name='Обо мне', null=True, blank=True, default='', max_length=150)
9      avatar = models.ImageField(verbose_name='Фото', blank=True, upload_to=get_timestamp_path_user)
10
11  USERNAME_FIELD = 'email'
12  REQUIRED_FIELDS = []
13
14  class Meta:
15      verbose_name_plural = 'Участники'
16      verbose_name = 'Участник'
17      ordering = ['last_name']
18
19  def __str__(self):
20      return f'Участник {self.first_name} {self.last_name}: {self.email}'
```

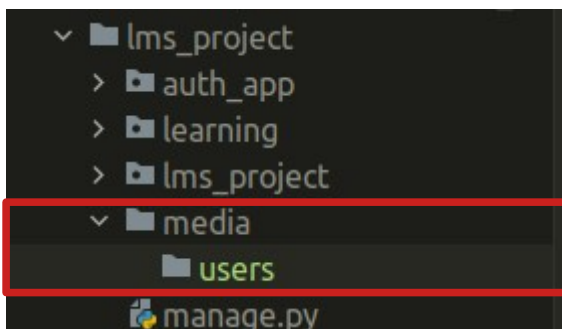
Что же здесь происходит?

Мы объявляем кастомный класс User, наследуясь от AbstractUser. Далее мы создаём три недостающих поля: birthday, description и avatar.

Атрибут класса `birthday` будет объектом класса поля `DateField`, которому мы задаём псевдоним и говорим, что он будет обязательным к заполнению, присваивая атрибуту `blank` значение `False`.

Атрибут класса `description` является объектом класса поля `TextField`. Мы будем использовать именно это поле, так как изначально неизвестно, какую длину оно может иметь. Однако мы ограничим его с помощью атрибута `max_length`.

Атрибут класса `avatar` является объектом класса `ImageField`. Он представляет из себя поле, содержащее строковое представление пути, куда загруженный пользователем графический файл сохранится. Атрибут `upload_to` указывает путь, куда файлы будут сохраняться. Нам нужно создать директорию для загруженных файлов в корневой папке проекта. Она будет иметь имя `media`. Конкретно для хранения аватарок пользователей мы создадим отдельную директорию в папке `media` под названием `users`.



Мы будем менять имя загруженного файла. К имени файла в начало мы будем добавлять, например, временную метку. Для этого создадим отдельный файл под именем `functions.py` и затем объявим функцию `get_timestamp_path_user`, параметрами которой будут текущая запись модели и начальное имя выгруженного файла. Эта функция возвратит конечный путь для сохранения файла.

```
1 from datetime import datetime
2 from os.path import splitext
3
4
5 def get_timestamp_path_user(instance, filename):
6     return f'users/{datetime.now().timestamp()}{splitext(filename)[1]}'
```

Нам нужно указать в настройках проекта директорию, в которой будут лежать выгруженные файлы. Сделать мы это можем, определив параметры `MEDIA_ROOT` и `MEDIA_URL`.

`MEDIA_ROOT` — это полный путь к папке, содержащей выгруженные файлы.

`MEDIA_URL` — префикс в URL, позволяющий Django понять, что выгруженный файл необходимо передать системе выгруженных файлов для последующей обработки.

```
126 # Media files
127 MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
128 MEDIA_URL = '/media/'
```

Также неявно этот тип поля имеет атрибут `allow_empty_file`, позволяющий разрешить / запретить пользователю загрузку файлов с нулевым объёмом. По умолчанию, он равен `False`. Помимо этого мы можем добавить валидатор `FileExtensionValidator`, который будет разрешать загрузку файлов только с определённым расширением. Он будет иметь следующие атрибуты:

- `allowed_extensions` — список, элементы которого представляют имена расширений графических файлов без начальной точки
- `error_messages` — сообщение в виде строки в случае попытки загрузки невалидного типа файла
- `code` — код ошибки

Перейдём к следующим параметрам: `USERNAME_FIELD` и `REQUIRED_FIELDS`.

`USERNAME_FIELD` позволяет указать поле, которое будет использоваться для авторизации (т. е. логин).

`REQUIRED_FIELDS` представляет собой список имён полей для создания суперпользователя.

Во вложенном классе Meta мы перечисляем свойства самой таблицы:

- **verbose_name_plural** — имя записи модели, понятное человеку, в множественном числе
- **verbose_name** — имя записи модели, понятное человеку, в единственном числе
- **ordering** — порядок сортировки. Мы сделаем сортировку по фамилии по возрастанию. Если мы хотим сделать по убыванию, то к имени поля необходимо добавить знак «-».

И, наконец, переопределим метод **__str__**. Он позволяет нам узнать информацию о текущем объекте. По умолчанию, Django возвращает номер объекта в БД.

Перейдём к созданию модели **Course**. Уже её нам придётся создавать с нуля. Следующие модели мы будем создавать в файле **models.py** приложения **learning**.

```
25 class Course(models.Model):
26     title = models.CharField(verbose_name='Название курса', max_length=30, unique=True)
27     author = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.PROTECT, verbose_name='Автор курса')
28     description = models.TextField(verbose_name='Описание курса', max_length=200)
29     start_date = models.DateField(verbose_name='Старт курса')
30     duration = models.PositiveIntegerField(verbose_name='Продолжительность')
31     price = models.PositiveIntegerField(verbose_name='Цена', blank=True)
32     count_lessons = models.PositiveIntegerField(verbose_name='Кол-во уроков')
33
34     class Meta:
35         verbose_name_plural = 'Курсы'
36         verbose_name = 'Курс'
37         ordering = ['title']
38
39     def __str__(self):
40         return f'{self.title}: Старт {self.start_date}'
```

Поле **title** является объектом класса поля CharField, ограничим его 30 символами и сделаем уникальным.

Поле **description** — объект класса TextField, ограничим его 200 символами.

Поле **author** представляет собой связь с пользовательской моделью типа один-ко-многим, так как один автор может иметь много курсов. Эту связь реализует поле **ForeignKey**. У него есть следующие атрибуты:

- **имя связываемой модели**, мы его получаем из ранее указанного параметра в файле настроек. Параметры файла настроек получаем путём импорта модуля **django.conf**

```
2 from django.conf import settings
```

- **on_delete** — задаёт поведение при попытке удаления записи первичной модели, т. е. в данном случае User. В свою очередь он может иметь следующие значения:
 - **PROTECT** — запрещает удалять запись первичной модели, выбрасывая при этом исключение ProtectedError
 - **CASCADE** — удаляет все записи вторичной модели, в данном случае Course.
 - **SETNULL** — заменит поле внешнего ключа на null только в том случае, если в этом поле задан атрибут null в значение True
 - **SETDEFAULT** - заменит поле внешнего ключа на значение по умолчанию, указанное в атрибуте default.

Поле **start_date** является полем типа DateField и представляет из себя строку в виде даты.

Поля **duration, price, count_lessons** являются объектами класса типа поля **PositiveIntegerField**. Тем самым мы вводим дополнительную проверку на положительность числа на уровне БД. Поле price является необязательным к заполнению, поскольку автор может решить выложить курс в свободный доступ (бесплатно).

В классе **Meta** мы также указываем свойства таблицы, задаём сортировку по полю `title` по возрастанию. И в конце перегружаем метод `__str__` для вывода информации о текущей записи.

Теперь создадим модель **Lesson**.

```
43 class Lesson(models.Model):
44     course = models.ForeignKey(Course, on_delete=models.CASCADE, verbose_name='Курс')
45     name = models.CharField(verbose_name='Название урока', max_length=25, unique=True)
46     preview = models.TextField(verbose_name='Описание урока', max_length=100)
47
48     class Meta:
49         verbose_name_plural = 'Уроки'
50         verbose_name = 'Урок'
51         ordering = ['course']
```

В этой модели поле **course** также имеет связь один-ко-многим, так как в курсе не может быть одного урока. В данном поле мы задаём каскадное удаление, тем самым при удалении курса будут удалены и все уроки, к нему привязанные.

Символьное поле **name** является уникальным. Сортировка производится по номеру курса. Логика установки свойств остальных полей уже должна быть понятна, поэтому перейдём к реализации последней модели.

Класс модели **Tracking** будет выглядеть следующим образом:

```
53 class Tracking(models.Model):
54     lesson = models.ForeignKey(Lesson, on_delete=models.PROTECT, verbose_name='Урок')
55     user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE, verbose_name='Ученик')
56     passed = models.BooleanField(default=None, verbose_name='Пройден?')
57
58     class Meta:
59         ordering = ['-id']
```

Поле **lesson** имеет связь один-ко-многим, так как здесь будет храниться вся статистика по всем курсам для всех учеников.

Поле **user** имеет ту же связь, потому что ученик может проходить одновременно несколько курсов. При удалении аккаунта происходит каскадное удаление его истории прохождения курсов.

Поле `passed` представляет собой BooleanField, по умолчанию равное None, потому что False станет только после завершения срока сдачи домашнего задания.

Сортировка в данном случае происходит по полю `user` по убыванию.

Миграции модели

Для конвертации текущего python-кода для последующей миграции в БД нам необходимо зафиксировать изменения во всём файле `models.py`

Для этого воспользуемся командой: `python manage.py makemigrations auth_app`

Запуск завершился неудачей с выводом следующих ошибок:

```
SystemCheckError: System check identified some issues:

ERRORS:
auth_app.User.avatar: (fields.E210) Cannot use ImageField because Pillow is not installed.
    HINT: Get Pillow at https://pypi.org/project/Pillow/ or run command "python -m pip install Pillow".
auth_app.User: (auth.E003) 'User.email' must be unique because it is named as the 'USERNAME_FIELD'.
```

Первая ошибка связана с тем, что у нас не установлена библиотека для обработки выгружаемых на сервер картинок Pillow.

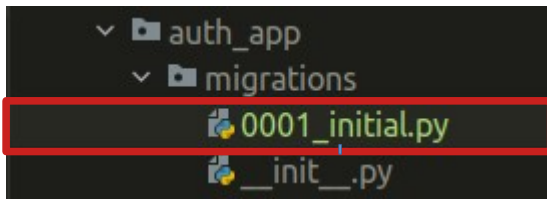
Также мы переопределили идентификатор в таблице, а в Django идентификатор должен быть уникальным значением. Поэтому добавим это поле в модель, указав соответствующие атрибуты:

```
7 class User(AbstractUser):
8     email = models.EmailField(unique=True, verbose_name='Email')
```

Далее введём повторно команду для формирования миграции.

```
Terminal: Local x + v
(venv) codeby@django:~/Desktop/lms_project/lms_project$ python manage.py makemigrations auth_app
Migrations for 'auth_app':
  auth_app/migrations/0001_initial.py
    - Create model User
    - Create model Course
    - Create model Lesson
    - Create model Tracking
(venv) codeby@django:~/Desktop/lms_project/lms_project$
```

Мы видим, что изменения в моделях были успешно зафиксированы и в папке миграций появился следующий файл `0001_initial.py`.



Этот файл содержит все зависимости миграций, необходимые операции, в данном случае создание моделей с их именем списком указанных полей, свойствами. Для создания миграции используется класс `Migration` из модуля `django.db.migrations`.

Небольшой фрагмент приведён ниже. Данный файл может иметь очень большой объём в зависимости от количества полей и их свойств.

```
class Migration(migrations.Migration):

    initial = True

    dependencies = [
        ('auth', '0012_alter_user_first_name_max_length'),
    ]

    operations = [
        migrations.CreateModel(
            name='User',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('password', models.CharField(max_length=128, verbose_name='password')),
                ('last_login', models.DateTimeField(blank=True, null=True, verbose_name='last login')),
                ('is_superuser', models.BooleanField(default=False, help_text='Designates that this user has all permissions without explicitly assigning them', verbose_name='is_superuser')),
                ('username', models.CharField(error_messages={'unique': 'A user with that username already exists'}, max_length=150, verbose_name='username')),
                ('first_name', models.CharField(blank=True, max_length=150, verbose_name='first name')),
                ('last_name', models.CharField(blank=True, max_length=150, verbose_name='last name')),
                ('is_staff', models.BooleanField(default=False, help_text='Designates whether the user can log in and administer the site', verbose_name='is staff')),
            ],
        ),
    ]
```

Теперь пришло время провести свою первую миграцию. Для этого введём команду: `python manage.py migrate`

Результатом выполнения данной команды будет вывод списка завершённых миграций.


```
(venv) codeby@django:~/Desktop/lms_project/lms_project$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, auth_app, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0001_initial... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying auth_app.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying sessions.0001_initial... OK
(venv) codeby@django:~/Desktop/lms_project/lms_project$
```

При первой миграции Django создаёт и свои таблицы для различных целей: для хранения миграций, сессий, разрешений и групп пользователей, в общем миграции всех приложений, указанных в **INSTALLED_APPS**.

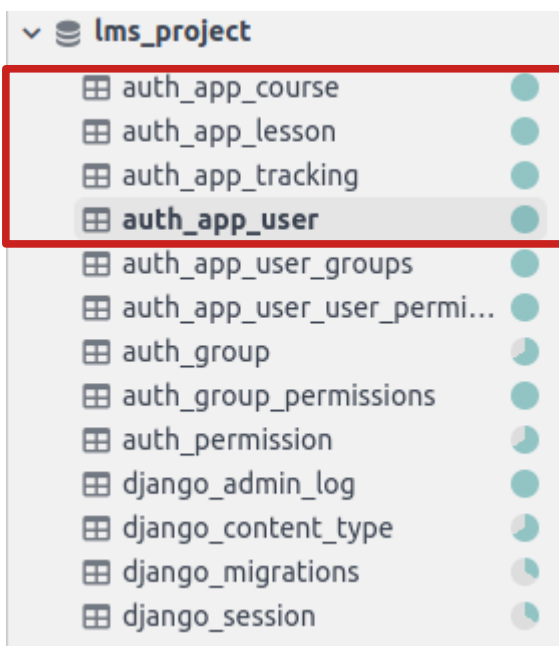
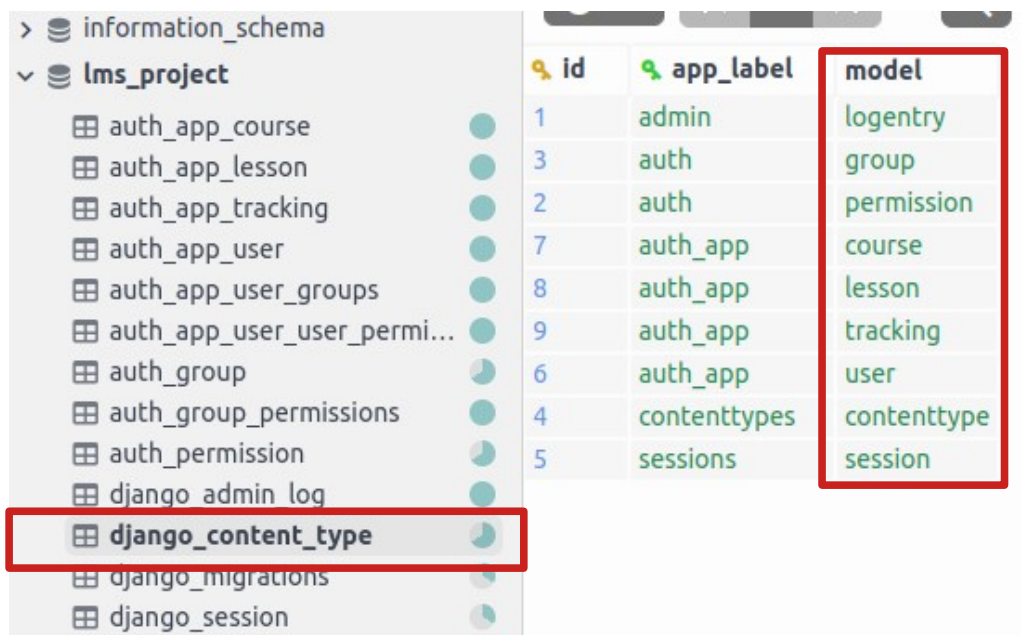


Таблица `django_content_type` позволяет нам узнать о всех моделях, зарегистрированных в нашем проекте.



id	app_label	model
1	admin	logentry
3	auth	group
2	auth	permission
7	auth_app	course
8	auth_app	lesson
9	auth_app	tracking
6	auth_app	user
4	contenttypes	contenttype
5	sessions	session

В ходе данного курса для облегчения мониторинга выполняемых операций над базой данных мы будем использовать клиентское приложение для взаимодействия с MySQL [Antares](#). Вы можете его установить в магазине приложений Ubuntu Software. Также разрешается установить любой альтернативный клиент для данной БД. Для подключения потребуется лишь имя пользователя и пароль, который Вы создали в прошлом уроке.

В следующем уроке мы познакомимся с интерфейсом административной панели Django, научимся регистрировать созданные модели и управлять ими.

📌 Домашнее задание

- ✓ Реализовать классы моделей Вашего приложения с учётом имеющихся ограничений (constraints)
- ✓ Реализовать кастомную пользовательскую модель
- ✓ Произвести миграцию в БД
- ✓ Установить клиент MySQL