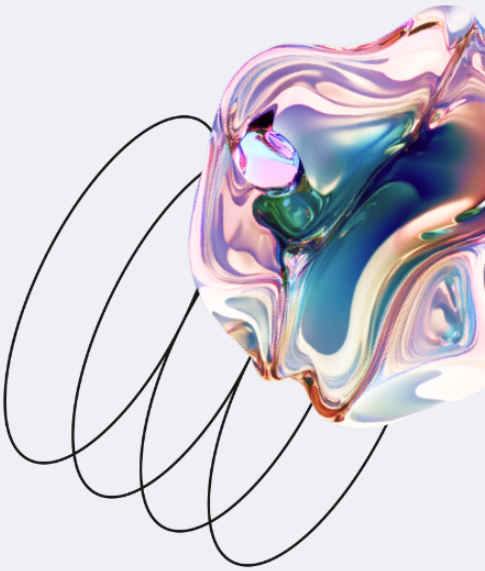
GeekBrains



Лекция 2

Работа с моделями

На этой лекции мы

1. Узнаем о моделях Django
2. Разберемся в создании моделей
3. Изучим миграции
4. Узнаем о создании собственных команд
5. Изучим работу с моделями данных, CRUD

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Узнали о фреймворке Django
2. Разобрались в его установке и настройке для первого запуска
3. Изучили структур проекта и работу с ним
4. Узнали о приложениях как частях проекта
5. Изучили настройки логирования в Django

План лекции

Введение в модели Django

Django - это популярный веб-фреймворк, который позволяет разрабатывать мощные и масштабируемые веб-приложения. Одной из ключевых функций Django являются модели, которые обеспечивают удобный способ работы с базами данных.

Модели в Django - это классы Python, которые определяют структуру таблиц базы данных. Каждый атрибут класса соответствует полю таблицы, а экземпляры класса представляют записи в таблице. Django использует ORM (Object-Relational Mapping), чтобы автоматически создавать SQL-запросы для работы с базой данных.

Зачем нужны модели?

Модели в Django обеспечивают удобный способ работы с базами данных. Они позволяют создавать, изменять и удалять записи в базе данных без необходимости написания SQL-запросов. Кроме того, модели позволяют описывать отношения между таблицами, что делает работу с базой данных еще более удобной и эффективной.

Примеры использования моделей в Django могут включать создание блогов, интернет-магазинов, социальных сетей и других приложений, которые требуют хранения и обработки большого количества данных.
Далее мы рассмотрим, как создавать и использовать модели в Django, а также как работать с данными в базе данных.

Определение моделей

Рассмотрим, как определять модели в Django. Для начала нам нужно создать новый проект Django (или продолжить работу в старом учебном проекте). Далее создадим новое приложение внутри проекта. Подобным мы занимались на прошлом занятии Напомним, что это можно сделать с помощью следующих команд:

```
django-admin startproject myproject
cd myproject
python manage.py startapp myapp2
```

Здесь мы создаем проект с именем "myproject" и приложение с именем "myapp2". После этого мы можем приступить к созданию модели.

Внимание! Для каждого занятия будет создаваться новое приложение: myapp2, myapp3 и т.д. В реальной разработке приложения делят проект на логические блоки, а не на итерации разработки.

Важно! Обязательно подключите ваше приложение в настройках проекта.

```
INSTALLED_APPS = [
    ...
    'myapp',
    'myapp2',
]
```

Создание модели

Для создания модели в Django мы должны перейти в файл models.py внутри вашего приложения и определить класс Python, который будет наследоваться от базового класса "models.Model".

Внимание! Если ваш проект состоит из нескольких приложений, создавать модели нужно в том приложении, которое явно подходит для работы с создаваемыми данными. В приложении “Пользователь” модель пользователя, в приложении “Блог” модель статьи и т.д.

Например, вот как можно определить модель для хранения информации о пользователях:

```
from django.db import models


class User(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField()
    password = models.CharField(max_length=100)
    age = models.IntegerField()
```

Здесь мы создаем класс "User", который наследуется от "models.Model". Мы определяем несколько полей модели: "name", "email", "password" и "age". Каждое поле имеет свой тип данных: "CharField" для строк, "EmailField" для электронных адресов и "IntegerField" для целых чисел.

Поля модели и их типы

Теперь давайте рассмотрим каждое поле более подробно:

- CharField - это поле для хранения строк. Мы указываем максимальную длину строки с помощью аргумента "max_length".
- EmailField - это поле для хранения электронных адресов. Django проверяет, что введенный адрес имеет правильный формат.
- IntegerField - это поле для хранения целых чисел.

Кроме того, у каждого поля есть несколько параметров, которые мы можем использовать для настройки поведения поля. Например, мы можем указать, что поле "name" обязательно для заполнения, используя параметр "blank=False".

Наиболее часто используемые поля

На самом деле в Django более трёх полей, которые мы использовали при создании пользователя. Кратко рассмотрим десяток самых частых полей.

1. CharField - поле для хранения строковых данных. Параметры: max_length (максимальная длина строки), blank (может ли поле быть

пустым), null (может ли поле содержать значение Null), default (значение по умолчанию).

2. IntegerField - поле для хранения целочисленных данных. Параметры: blank, null, default.
3. TextField - поле для хранения текстовых данных большой длины. Параметры: blank, null, default.
4. BooleanField - поле для хранения логических значений (True/False). Параметры: blank, null, default.
5. DateField - поле для хранения даты. Параметры: auto_now (автоматически устанавливать текущую дату при создании объекта), auto_now_add (автоматически устанавливать текущую дату при добавлении объекта в базу данных), blank, null, default.
6. DateTimeField - поле для хранения даты и времени. Параметры: auto_now, auto_now_add, blank, null, default.
7. ForeignKey - поле для связи с другой моделью. Параметры: to (имя модели, с которой устанавливается связь), on_delete (действие при удалении связанного объекта), related_name (имя обратной связи).
8. ManyToManyField - поле для связи с другой моделью в отношении "многие-ко-многим". Параметры: to, related_name.
9. DecimalField - поле для хранения десятичных чисел. Параметры: max_digits (максимальное количество цифр), decimal_places (количество знаков после запятой), blank, null, default.
10. EmailField - поле для хранения электронной почты. Параметры: max_length, blank, null, default.

Далее в рамках лекции и курса мы поработаем с большинством из них на практике. Полный список всех полей Django доступен по ссылке: https://docs.djangoproject.com/en/4.2/ref/models/fields/#model-field-types

Несколько примеров моделей

Рассмотрим ещё пару примеров моделей

```
class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=8, decimal_places=2)
    description = models.TextField()
    image = models.ImageField(upload_to='products/')
```

```
class Order(models.Model):
    customer = models.ForeignKey(User, on_delete=models.CASCADE)
    products = models.ManyToManyField(Product)
    date_ordered = models.DateTimeField(auto_now_add=True)
    total_price = models.DecimalField(max_digits=8, decimal_places=2)
```

Здесь мы создаем модели для хранения информации о продуктах и заказах. В модели "Product" мы определяем поля для хранения названия продукта, цены, описания и изображения. В модели "Order" мы определяем поля для хранения информации о заказчике, списке продуктов, дате заказа и общей стоимости заказа. Обратите внимание на использование ForeignKey и ManyToManyField для определения отношений между таблицами.

- `customer = models.ForeignKey(User, on_delete=models.CASCADE)` - каждый заказ делает конкретный пользователь. У одного пользователя может быть несколько заказов, но заказ числится за одним пользователем.
- `products = models.ManyToManyField(Product)` - заказа может содержать несколько разных продуктов. А продукт может входит в состав нескольких разных заказов.

Внимание! Для использования в моделях Django поля ImageField необходимо установить дополнительный модуль Pillow. Выполните команду внутри виртуального окружения проекта:

```
pip install Pillow
```

Данная библиотека нужна для работы Python с изображениями.

В этой части лекции мы рассмотрели, как создавать модели в Django и как определять поля моделей. Модели представляют собой удобный способ работы с базами данных в Django, который позволяет создавать, изменять и удалять записи в базе данных без необходимости написания SQL-запросов.

Миграции

В Django **миграции** представляют собой автоматически сгенерированные скрипты для изменения структуры базы данных в соответствии с изменениями в моделях приложения. Миграции позволяют легко и безопасно изменять базу данных, не теряя данные.

Как создать миграции для моделей

Для создания миграций для моделей Django используется команда "makemigrations". Например, чтобы создать миграции для модели "User", мы должны выполнить следующую команду:

```
python manage.py makemigrations myapp2
```

Здесь "myapp2" - это имя нашего приложения, в котором определена модель "User". Команда "makemigrations" анализирует все изменения в моделях приложения и создает миграционный файл, который содержит инструкции для изменения базы данных.

Внимание! Если опустить имя приложения, команда `python manage.py makemigrations` попытается найти изменения во всех приложениях проекта и создать миграции для каждого из них.

Важно! Перед запуском команды убедитесь, что ваше приложение добавлено в список INSTALLED_APPS файла settings.py вашего проекта:

```
INSTALLED_APPS = [
    ...
    "myapp2",
    ...
]
```

Если заглянуть в каталог /migrations вашего приложения, увидите новый файл вида 0001_initial.py. Содержимое файла сгенерировано автоматически и практически никогда не требует ручных правок.

Превращение модели пользователя в миграцию

В файле models.py мы создали следующую модель пользователя:

```
class User(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField()
    password = models.CharField(max_length=100)
    age = models.IntegerField()
```

После миграции она превратилась в следующий набор кода:

```
        migrations.CreateModel(
            name='User',
            fields=[
                ('id', models.BigAutoField(auto_created=True,
primary_key=True, serialize=False, verbose_name='ID')),
                ('name', models.CharField(max_length=100)),
                ('email', models.EmailField(max_length=254)),
                ('password', models.CharField(max_length=100)),
                ('age', models.IntegerField()),
            ],
        ),
```

В базу данных автоматически будет добавлено поле id как первичный ключ. Также для почты автоматически введено ограничение на длину поля. Django делает часть преобразований самостоятельно, там где они подразумеваются.

Внимание! Создание миграций не изменяет таблицы базы данных. Это лишь подготовка к изменениям.

Применение миграций

Вспомните запуск сервера

```
>python manage.py runserver
...
You have 19 unapplied migration(s). Your project may not work properly
until you apply the migrations for app(s): admin, auth, contenttypes,
myapp2, sessions.
Run 'python manage.py migrate' to apply them.
```

Теперь Django указывает на 19 неприменённых миграций, вместо 18 ранее. В список добавилось приложение myapp2 после команды makemigrations.
После создания миграционного файла мы должны применить его к базе данных с помощью команды "migrate":

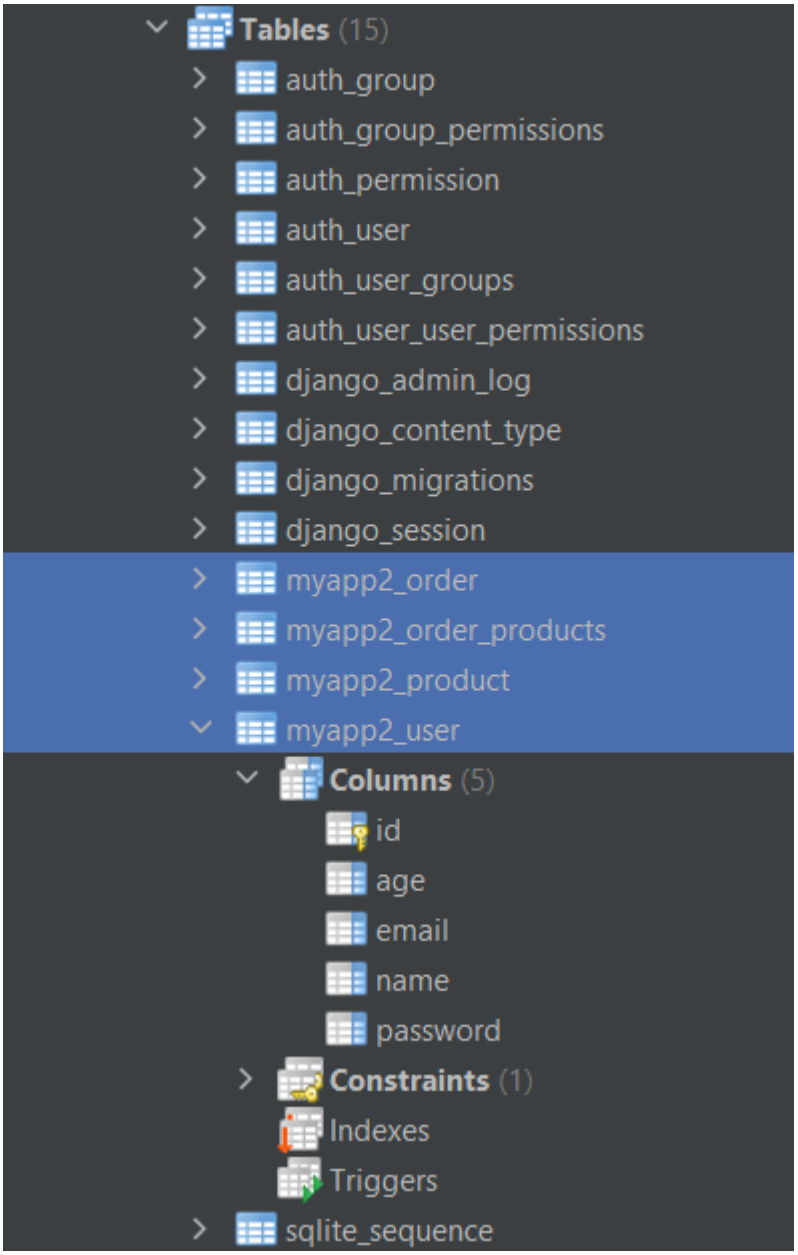
```
python manage.py migrate
```

Команда "migrate" выполняет все накопленные миграции в порядке их создания и применяет изменения к базе данных. В первый раз получим нечто подобное:

```
>python manage.py migrate
```

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, myapp2, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.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 contenttypes.0002_remove_content_type_name... 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 myapp2.0001_initial... OK
  Applying sessions.0001_initial... OK
```

База данных при этом наполнится таблицами, включая пользователя, которого мы создали ранее. А если вы оставили модели товары и заказы при создании миграций, эти таблицы также будут созданы в БД



Важно отметить, что при каждом изменении модели необходимо создать новые миграции для этих изменений. Затем применить их к базе данных, мигрировать в базу. Это позволяет обновлять базу данных без потери данных и сохранять целостность базы данных.

Пара слов о создании собственных команд manage.py

Прежде чем двигаться дальше, рассмотрим возможности Django по созданию собственных команд. Они пригодятся при тестировании работы с моделями

данных без создания представлений, добавления маршрутов и отрисовки шаблонов.

Создаём структуру каталогов

Для начала в каталоге приложения необходимо создать следующие вложенные друг в друга каталоги: management/ и commands/
Получим следующую структуру

```
myproject/
    manage.py
    myapp2/
        __init__.py
        management/
            __init__.py
            commands/
                __init__.py
                my_command.py
        ...
    ...
```

Создаём файл с кодом команды

При этом простейшее содержимое файла должно быть следующим Назовём файл my_command.py:

```
from django.core.management.base import BaseCommand


class Command(BaseCommand):
    help = "Print 'Hello world!' to output."

    def handle(self, *args, **kwargs):
        self.stdout.write('Hello world!')
```

Создаём класс Command как дочерний для BaseCommand. Переменная help выведет справку по работе команды. А метод handle отработает при вызове команды в консоли.

Внимание! Вместо привычного print необходимо использовать self.stdout.write для печати информации в стандартный поток вывода - консоль.

Вызываем команду

Файл my_command.py можно запускать из терминала командой

```
python manage.py my_comand
```

В результате мы увидим текст “Hello world!” в консоли.
Отлично. Теперь мы можем вернуться к работе с базой данных.

Работа с данными в моделях

В Django работа с данными в моделях осуществляется через объекты моделей, которые представляют отдельные записи в базе данных. Рассмотрим основные операции с объектами модели.
CRUD - акроним, обозначающий четыре базовые функции, используемые при работе с базами данных: создание (create), чтение (read), модификация (update), удаление (delete).

Создание объектов модели, create

Для создания нового объекта модели необходимо создать экземпляр класса модели и заполнить его поля значениями. Например, чтобы создать нового пользователя, мы можем использовать следующий код в файле myapp2/management/commands/create_user.py:

```
from django.core.management.base import BaseCommand
from myapp2.models import User


class Command(BaseCommand):
    help = "Create user."

    def handle(self, *args, **kwargs):
        user = User(name='John', email='john@example.com',
password='secret', age=25)
        user.save()
        self.stdout.write(f'{user}')
```

Здесь мы создаем новый объект модели "User" с заданными значениями полей и сохраняем его в базе данных с помощью метода "save()". Далее выводим на печать сохранённого пользователя.
Если заглянуть в базу данных, таблица myapp2_user получит новую запись.

Добавляем читаемое представление объекта в модель

После запуска команды

```
python manage.py create_user
```

вы увидите строку вида:

```
User object (1)
```

Не очень удобно для работы с пользователем. Для повышения читаемости откроем файл models.py с моделью User и добавим дандер метод __str__. Получим следующий код:

```
...
class User(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField()
    password = models.CharField(max_length=100)
    age = models.IntegerField()

    def __str__(self):
        return f'Username: {self.name}, email: {self.email}, age:
{self.age}'
...
```

Запускаем команду повторно и видим:

```
>python manage.py create_user
Username: John, email: john@example.com, age: 25
```

Внимание! Если вы проверите содержимое таблицы myapp2_user в базе данных, обнаружите там несколько одинаковых записей. Отличаться они будут только порядковым номером в поле id. Каждый запуск команды create_user создаёт новую запись в БД.

Прежде чем двигаться дальше, можем создать ещё несколько тестовых пользователей, предварительно изменив их данные в коде:

```
...
user = User(name='Neo', email='neo@example.com', password='secret',
age=58)
...
```

Получение объектов модели из базы данных, read

Для получения объектов модели из базы данных можно использовать методы "all()" и "get()" класса модели. Метод "all()" возвращает все объекты модели, а метод "get()" возвращает один объект, соответствующий заданным условиям.
Например, чтобы получить всех пользователей из базы данных, мы можем использовать следующий код в файле myapp2/management/commands/get_all_users.py:

```
from django.core.management.base import BaseCommand
from myapp2.models import User


class Command(BaseCommand):
    help = "Get all users."

    def handle(self, *args, **kwargs):
        users = User.objects.all()
        self.stdout.write(f'{users}')
```

А чтобы получить пользователя по его ID, мы можем использовать следующий код в файле myapp2/management/commands/get_user.py:

```
from django.core.management.base import BaseCommand
from myapp2.models import User


class Command(BaseCommand):
    help = "Get user by id."

    def add_arguments(self, parser):
        parser.add_argument('id', type=int, help='User ID')

    def handle(self, *args, **kwargs):
        id = kwargs['id']
        user = User.objects.get(id=id)
        self.stdout.write(f'{user}')
```

Метод add_arguments позволяет парсить командную строку. Мы получаем значение целого типа и сохраняем его по ключу id. Теперь обработчик handler может получить к идентификатору доступ через ключ словаря kwargs.
Запустим команду

```
python manage.py get_user 2
```

В результате в консоли увидим информацию о нашем пользователе

```
Username: John, email: john@example.com, age: 25
```

Отлично работает. Но если капнуть глубже, есть нюанс.

Получение объекта или None

Если мы запустим прошлый пример с несуществующим в БД идентификатором, получим ошибку: `myapp2.models.User.DoesNotExist: User matching query does not exist`.

Исправим наш код:

```
from django.core.management.base import BaseCommand
from myapp2.models import User


class Command(BaseCommand):
    help = "Get user by id."

    def add_arguments(self, parser):
        parser.add_argument('pk', type=int, help='User ID')

    def handle(self, *args, **kwargs):
        pk = kwargs['pk']
        user = User.objects.filter(pk=pk).first()
        self.stdout.write(f'{user}')
```

Во-первых обратите внимание на замену id на pk. Так команда Django ушла от конфликта между именем переменной и встроенной в Python функцией id().
pk - primary key, первичный ключ в таблице, т.е. ID.
Также для получения пользователя заменили метод get на filter, а далее к результату применяем метод first().

- Если в базе несколько записей, вернёт одна, первая из результата запроса
- Если запись одна, first вернёт эту единственную запись
- Если совпадений не найдено, вместо ошибки вернём None

Рассмотрим работу с filter подробнее

Фильтрация объектов модели

Для фильтрации объектов модели по заданным условиям можно использовать метод "filter()" класса модели. Метод "filter()" возвращает все объекты модели, удовлетворяющие заданным условиям.

Например, чтобы получить всех пользователей старше <age> лет, мы можем использовать следующий код в файле myapp2/management/commands/get_user_age_greater.py:

```
from django.core.management.base import BaseCommand
from myapp2.models import User


class Command(BaseCommand):
    help = "Get user with age greater <age>."

    def add_arguments(self, parser):
        parser.add_argument('age', type=int, help='User age')

    def handle(self, *args, **kwargs):
        age = kwargs['age']
        user = User.objects.filter(age__gt=age)
        self.stdout.write(f'{user}')
```

Здесь мы используем оператор "__gt" для сравнения значения поля "age" с заданным значением.

Помимо оператора __gt существуют множество других. Перечислим некоторые часто используемые:

- exact - точное совпадение значения поля
- iexact - точное совпадение значения поля без учета регистра
- contains - значение поля содержит заданный подстроку
- icontains - значение поля содержит заданный подстроку без учета регистра
- in - значение поля находится в заданном списке значений
- gt - значение поля больше заданного значения
- gte - значение поля больше или равно заданному значению
- lt - значение поля меньше заданного значения
- lte - значение поля меньше или равно заданному значению
- startswith - значение поля начинается с заданной подстроки
- istartswith - значение поля начинается с заданной подстроки без учета регистра
- endswith - значение поля заканчивается на заданную подстроку
- iendswith - значение поля заканчивается на заданную подстроку без учета регистра
- range - значение поля находится в заданном диапазоне значений
- date - значение поля является датой, соответствующей заданной дате
- year - значение поля является годом, соответствующим заданному году

Как вы догадались, приставка i означает, что поиск будет производиться без учета регистра символов. Например, iexact найдет записи с точным совпадением значения поля, но не будет учитывать регистр символов при поиске.

Важно! Помните про двойное подчеркивание перед оператором. Например для поиска имён начинающихся с S будет использоваться запись вида: `name__startswith='S'`

Более подробно познакомиться с возможностью фильтров для методов filter(), exclude() и get() можно на официальном сайте https://docs.djangoproject.com/en/4.2/ref/models/querysets/#field-lookups

Изменение объектов модели, update

Для изменения объектов модели можно использовать методы поиска get() или filter() в сочетании с save() экземпляра класса модели. Например, чтобы изменить имя пользователя с заданным id, мы можем использовать следующий код в файле myapp2/management/commands/update_user.py::

```
from django.core.management.base import BaseCommand
from myapp2.models import User


class Command(BaseCommand):
    help = "Update user name by id."

    def add_arguments(self, parser):
        parser.add_argument('pk', type=int, help='User ID')
        parser.add_argument('name', type=str, help='User name')

    def handle(self, *args, **kwargs):
        pk = kwargs.get('pk')
        name = kwargs.get('name')
        user = User.objects.filter(pk=pk).first()
        user.name = name
        user.save()
        self.stdout.write(f'{user}')
```

Выполним команду

```
>python manage.py update_user 2 Smith
```

```
Username: Smith, email: john@example.com, age: 25
```

Здесь мы получаем пользователя с первичным ключом 2, изменяем его имя на "Smith" и сохраняем изменения в базе данных с помощью метода "save()".

Удаление объектов модели, delete

Для удаления объектов модели можно использовать методы delete() экземпляра класса модели. Если надо удалить пользователя, мы можем использовать следующий код в файле myapp2/management/commands/delete_user.py:

```
from django.core.management.base import BaseCommand
from myapp2.models import User


class Command(BaseCommand):
    help = "Delete user by id."

    def add_arguments(self, parser):
        parser.add_argument('pk', type=int, help='User ID')

    def handle(self, *args, **kwargs):
        pk = kwargs.get('pk')
        user = User.objects.filter(pk=pk).first()
        if user is not None:
            user.delete()
        self.stdout.write(f'{user}')
```

Здесь мы получаем пользователя по id и удаляем его из базы данных с помощью метода "delete()".

Внимание! Не стоит злоупотреблять методом delete в проектах. Возможно данные понадобятся в будущем. Логичнее добавить в модель поле is_deleted = BooleanField(). В случае “удаления” ставить в поле флаг True.

Дополнительные возможности моделей

Связи между моделями

Отношения между моделями в Django позволяют описывать связи между объектами разных моделей. Для этого используются поля моделей, такие как ForeignKey, ManyToManyField и OneToOneField.
Например, у нас есть модель Post и модель Author, и каждый пост может быть написан только одним автором, а автор может написать много постов. Для этого мы можем использовать поле ForeignKey в модели Post, которое будет ссылаться на модель Author.
Внесём код в models.py учебного приложения:

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

    def __str__(self):
        return f'Name: {self.name}, email: {self.email}'


class Post(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    author = models.ForeignKey(Author, on_delete=models.CASCADE)

    def __str__(self):
        return f'Title is {self.title}'
```

Здесь мы создаем модели Author и Post с полями "name", "email", "title", "content" и "author". Поле "author" в модели Post является ForeignKey, которое ссылается на модель Author.

Внимание! Не забываем про миграции после изменения файла models.py

```
>python manage.py makemigrations
Migrations for 'myapp2':
  myapp2\migrations\0002_author_post.py
    - Create model Author
    - Create model Post

>python manage.py migrate
Operations to perform:
```

```
  Apply all migrations: admin, auth, contenttypes, myapp2, sessions
Running migrations:
  Applying myapp2.0002_author_post... OK
```

Создание фейковых авторов и статей

Создадим файл myapp2/management/commands/fake_data.py. Он поможет заполнить базу тестовыми данными.

Внимание! Обычно тестовые данные используются при написании тестов. Django в этом случае создаёт тестовую базу и не затрагивают основную. Будьте внимательны, не сломайте базу данных из продакшена фейковыми данными или случайным удалением данных.

```
from django.core.management.base import BaseCommand
from myapp2.models import Author, Post


class Command(BaseCommand):
    help = "Generate fake authors and posts."

    def add_arguments(self, parser):
        parser.add_argument('count', type=int, help='User ID')

    def handle(self, *args, **kwargs):
        count = kwargs.get('count')
        for i in range(1, count + 1):
            author = Author(name=f'Name{i}', email=f'mail{i}@mail.ru')
            author.save()
            for j in range(1, count + 1):
                post = Post(title=f'Title{j}', content=f'Text from
{author.name} #{j} is bla bla bla many long text', author=author)
                post.save()
```

Запустив команду >python manage.py fake_data 10 получим 10 авторов и 100 статей.

Использование QuerySet для получения данных из базы данных

Для получения данных из базы данных мы можем использовать QuerySet, который позволяет выполнить различные операции над набором объектов

модели. Например, чтобы получить все посты, написанные определенным автором, мы можем использовать следующий код в файле myapp2/management/commands/get_post_by_author_id.py:

```
from django.core.management.base import BaseCommand
from myapp2.models import Author, Post


class Command(BaseCommand):
    help = "Get all posts by author id."

    def add_arguments(self, parser):
        parser.add_argument('pk', type=int, help='User ID')

    def handle(self, *args, **kwargs):
        pk = kwargs.get('pk')
        author = Author.objects.filter(pk=pk).first()
        if author is not None:
            posts = Post.objects.filter(author=author)
            intro = f'All posts of {author.name}\n'
            text = '\n'.join(post.content for post in posts)
            self.stdout.write(f'{intro}{text}')
```

Здесь мы получаем автора по его id и фильтруем все посты, чтобы получить только те, которые были написаны им.
А если нам не нужно имя автора в строке intro, можем изменить запрос к базе данных. Фильтруем посты по автору непосредственно из модели пост:

```
    ...
    def handle(self, *args, **kwargs):
        pk = kwargs.get('pk')
        posts = Post.objects.filter(author__pk=pk)
        intro = f'All posts\n'
        text = '\n'.join(post.content for post in posts)
        self.stdout.write(f'{intro}{text}')
```

Мы просим найти посты где у поля автор первичный ключ равен pk.

Создание пользовательских методов

Также мы можем создавать пользовательские методы и свойства для моделей, чтобы расширить их функциональность. Например, мы можем создать метод "get_summary" для модели "Post", который будет возвращать краткое описание поста. Внесём изменения в класс Post в файле models.py:

```
class Post(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    author = models.ForeignKey(Author, on_delete=models.CASCADE)

    def __str__(self):
        return f'Title is {self.title}'

    def get_summary(self):
        words = self.content.split()
        return f'{" ".join(words[:12])}...'
```

Здесь мы создаем метод "get_summary", который возвращает первые 12 слов контента поста и добавляет многоточие в конце.

```
...
text = '\n'.join(post.get_summary() for post in posts)
...
```

Вместо полного текста теперь можем получать первые несколько слов.

Django не ограничивает разработчика не методы внутри моделей. Подобный подход, когда данные и расчёты производятся в моделях более предпочтителен, чем расчёты внутри представлений по выгруженным из модели данным. Тем более не стоит переносить расчёты в представления, о которых мы будем подробно говорить на следующей лекции.

Вывод

На этой лекции мы:

1. Узнали о моделях Django
2. Разобрались в создании моделей
3. Изучили миграции
4. Узнали о создании собственных команд
5. Изучили работу с моделями данных, CRUD

Домашнее задание

1. Для закрепления материалов лекции попробуйте самостоятельно набрать и запустить демонстрируемые примеры.

2. *Загляните в официальную документацию Django и изучите дополнительные возможности работы с моделями.