**Урок 2: Работа с моделями**

**Модели в Django - это классы Python, которые определяют структуру таблиц**

**базы данных.**

**Активируем виртуальное окружение и переходим в каталог с проектом.**

.\venv\Scripts\activate

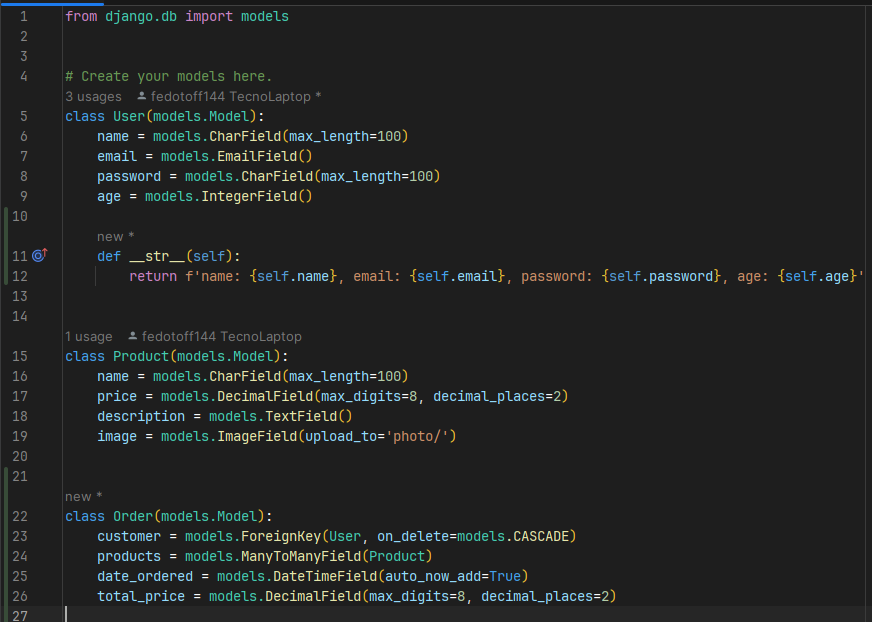
**Создаем новое приложение и добавляем его в recipes/settings.py**.

cd .\recipesbook\

python manage.py startapp userapp

**Создаем модель**

Переходим в файл userapp/models.py для создания модели.



Мы создали модель пользователя, модель продукта и модель заказа.

Внимане! Поле id не указываем потому что Django создаст его автоматически.

Внимание! После добавления поля image = models.ImageField(upload\_to='photo/') необходимо добавить пакет Pillow для запуска сервера.

pip install Pillow

Подробнее о полях по ссылке: <https://docs.djangoproject.com/en/4.2/ref/models/fields/#model-field-types>

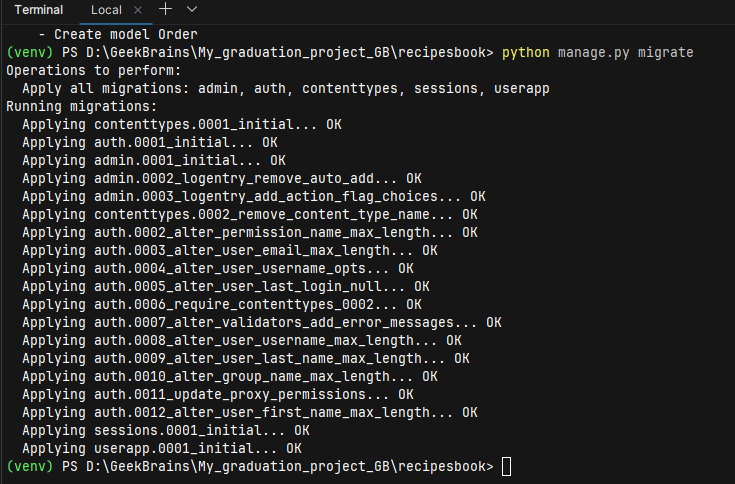
**После создания моделей, нам нужно подготовить их к переносу в БД (создать миграции) для нашего приложения userapp**

python manage.py makemigrations userapp

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

**Теперь нам надо применить миграции, то есть залить все структуры в БД**

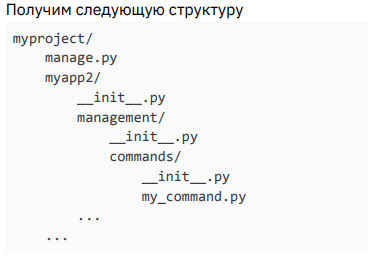
python manage.py migrate



База данных создана и теперь к ней можно подключиться

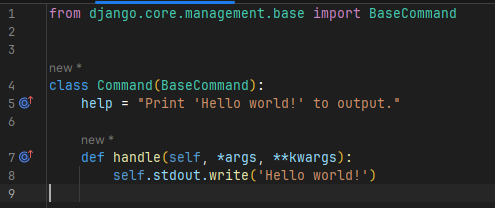
**Создание собственных команд manage.py**

Для начала нам надо создать структуру каталогов для команд.



Где my\_command.py имя нашей команды, которая будет делать определенные действия.

Пишем логику команды



**Запускаем нашу команду из терминала командой**

python manage.py my\_command

Посмотреть справку нашей команды, которая указана в строке help =….

python manage.py my\_command -h

**Работа с данными в моделях**

**Создание объектов модели CREATE**

Добавляем команду create\_user.py и добавляем её в пакет с собственными командами



В терминале вводим нашу команду для создания пользователя

python manage.py create\_user

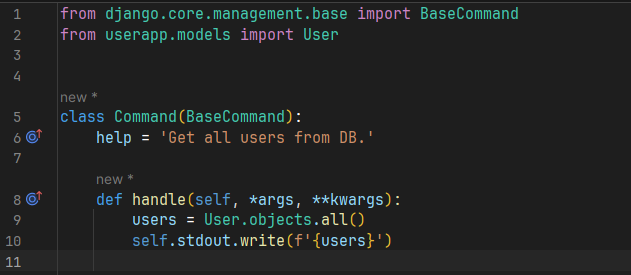
Подключаемся к нашей БД db.sqlite3 и видим добавленного пользователя. Также в имея дандер метод \_\_str\_\_ у нашей модели мы в терминале видим данные созданного пользователя



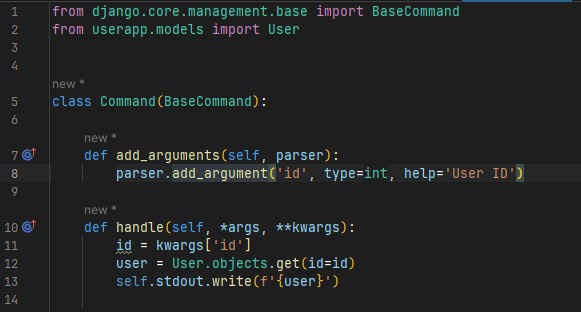
**Получение объектов модели из базы данных READ**

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.

Добавляем команду get\_all\_users.py и добавляем её в пакет с собственными командами (применение "all()")



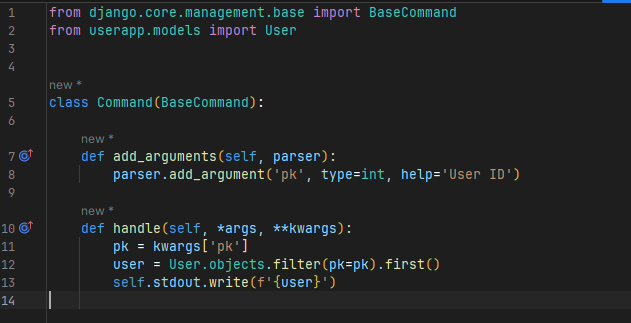
Добавляем команду get\_user\_by\_id.py и добавляем её в пакет с собственными командами командами (применение "get()")

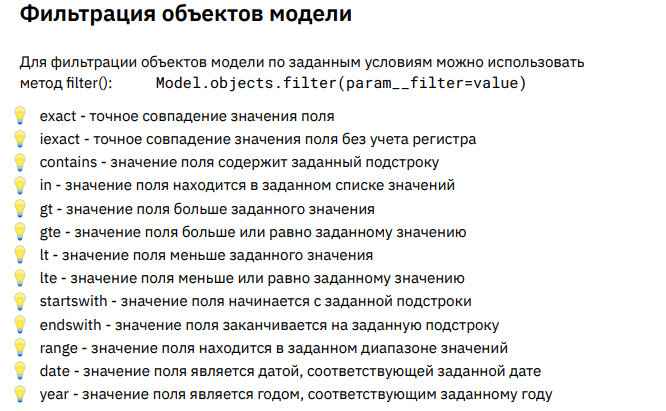




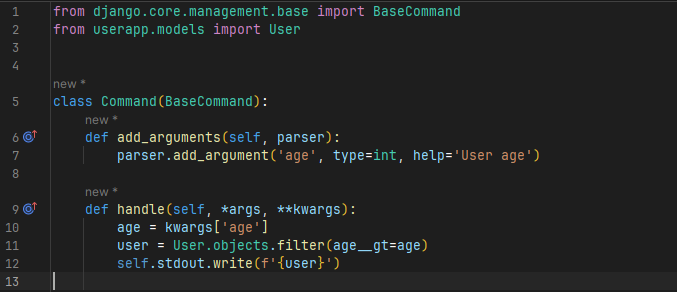
Внимание! При использовании get() можно поймать ошибку в случае если пользователя с указанным id нет в базе, поэтому принято использовать filter()

Добавляем команду get\_user\_through\_filter.py и добавляем её в пакет с собственными командами командами (применение "filter()")



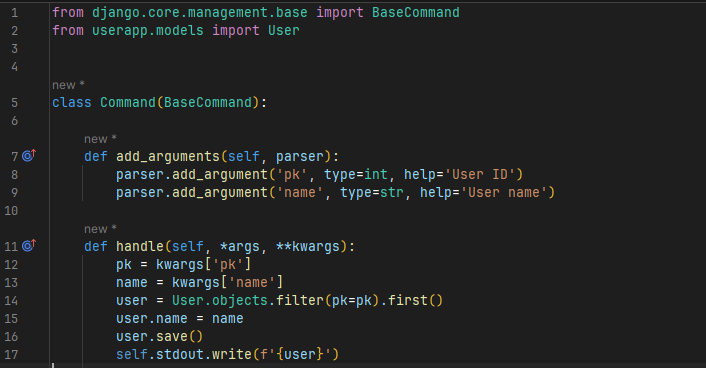
Дополнительная информация по filter() 🡪  


Пример:



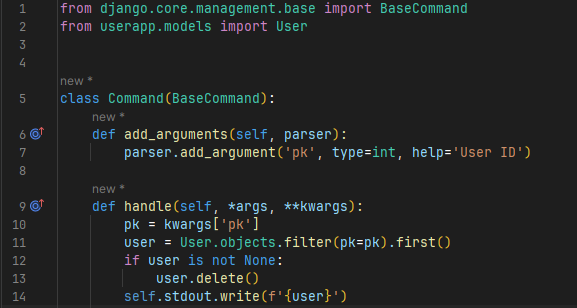
**Изменение объектов модели UPDATE**

Для изменения объектов модели можно использовать методы поиска get() или filter() в сочетании с save() экземпляра класса модели



**Удаление объектов модели DELETE**

Для удаления объектов модели можно использовать методы поиска get() или filter() в сочетании с delete() экземпляра класса модели



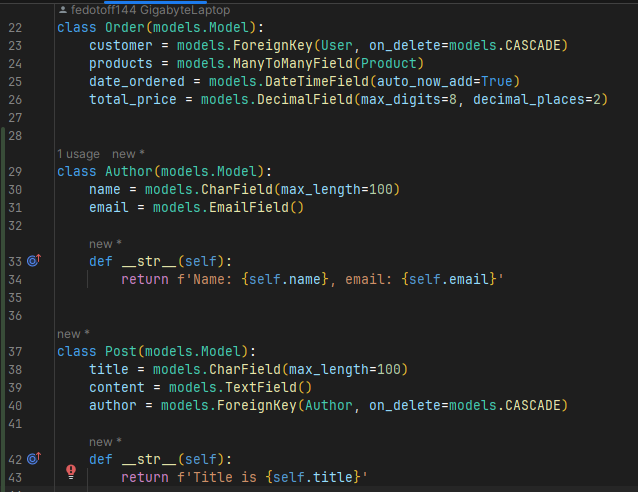
**Связи между моделями**

● ForeignKey - один ко многим

● ManyToManyField - многие ко многим

● OneToOneField - один к одному

Добавим модели в наш проект

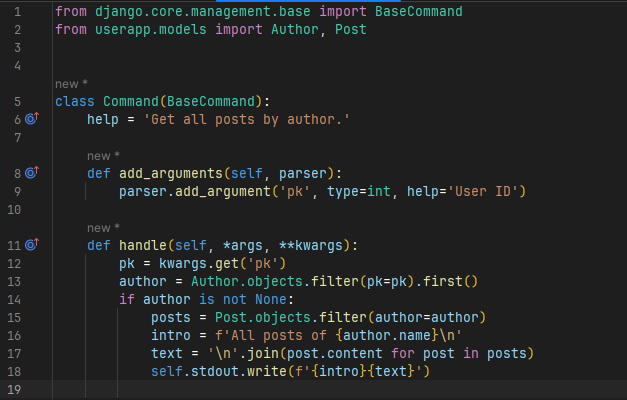


Внимание! После создания моделей обязательно нужно сделать миграции, иначе изменения не вступят в силу

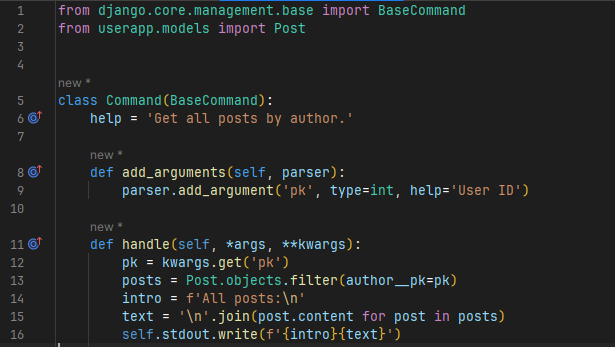
python manage.py makemigrations userapp

python manage.py migrate

Получаем все посты автора



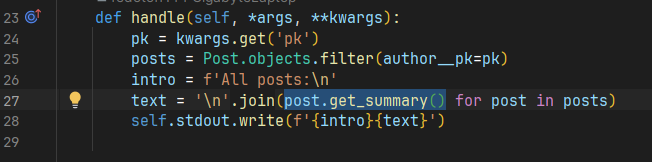
Или

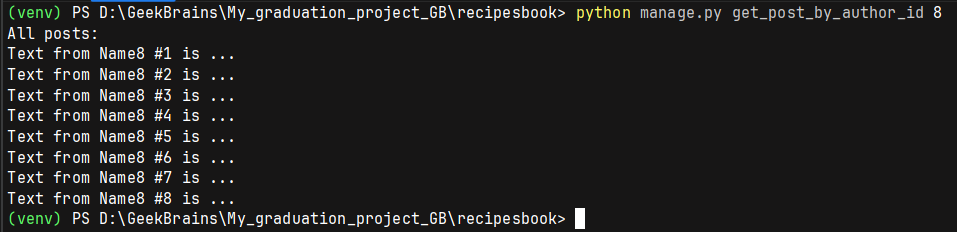


**Создание пользовательских методов**

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 usserapp\models.py:

После этого мы доработаем нашу команду get\_post\_by\_author\_id в папке userapp/management/commands/get\_post\_by\_author\_id.py



Изменения на скрине выделены и теперь мы получаем следующий вывод:  


**Так же существует class Meta в котором указывается дополнительная информация вроде сортировки или перечисления полей, которые будут использоваться. Пример на скрине, а дополнительная информация по ссылке: https://docs.djangoproject.com/en/5.0/ref/models/options/**

